从机器学习到深度学习

基于scikit-learn与TensorFlow的高效开发实战

刘长龙 / 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

VISIT…

LANZAROTE Caliente.COM

内 容 简 介

这是一本场景式的机器学习实践书，笔者努力做到“授人以渔，而非授人以鱼”。理论方面从人工智能（AI）与机器学习（ML）的基本要素讲起，逐步展开有监督学习、无监督学习、强化学习这三大类模型的应用场景与算法原理；实践方面通过金融预测、医疗诊断概率模型、月球登陆器、图像识别、写诗机器人、中国象棋博弈等案例启发读者将机器学习应用在各行各业里，其中后三个案例使用了深度学习技术。

本书试图用通俗的语言讲解涵盖算法模型的机器学习，主要内容包括机器学习通用概念、三个基本科学计算工具、有监督学习、聚类模型、降维模型、隐马尔可夫模型、贝叶斯网络、自然语言处理、深度学习、强化学习、模型迁移等。在深入浅出地解析模型与算法之后，介绍使用 Python 相关工具进行开发的方法、解析经典案例，使读者做到“能理解、能设计、能编码、能调试”，没有任何专业基础的读者在学习本书后也能够上手设计与开发机器学习产品。

本书内容深入浅出、实例典型，适合对机器学习感兴趣的产品设计、技术管理、数据分析、软件开发或学生读者。阅读本书既能了解当前工业界的主流机器学习与深度学习开发工具的使用方法，又能从战略方面掌握如何将人工智能技术应用到自己的企业与产品中。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

从机器学习到深度学习：基于 scikit-learn 与 TensorFlow 的高效开发实战 / 刘长龙著. —北京：电子工业出版社，2019.3
ISBN 978-7-121-35518-9

Ⅰ. ①从…　Ⅱ. ①刘…　Ⅲ. ①机器学习　Ⅳ.①TP181

中国版本图书馆 CIP 数据核字（2018）第 252504 号

责任编辑：董　英
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：32　字数：642 千字
版　　次：2019 年 3 月第 1 版
印　　次：2019 年 3 月第 1 次印刷
定　　价：99.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

谨以此书献给我家有无限活力的小朋友，

感谢他给予我的无穷动力！

前　　言

随着越来越多的人工智能技术的突破、市场上实际产品的研发与销售，机器学习成为了当下所有高科技企业在管理、技术、运维等所有层面被高度关注的技术领域。在所有企业的产品设计论坛、技术研讨会中，人工智能与机器学习总会成为大家关注的焦点。

但是，人工智能与机器学习所获得的关注与大家对该领域知识的了解却不成正比。以笔者自身的经历来说，在顶尖高科技企业的技术研讨与分享会中，很多次被问到：

机器学习与深度学习的区别是什么？

什么是有监督学习？

人工智能能不能应用在我们的产品中？

这些问题既可以用一两句话回答，也可以用一本书来阐述。无论如何，笔者能肯定的一点是：**大家对机器学习充满兴趣，但平时又忙于工作和生活，无暇系统地学习这方面的知识。**

机器学习作为一个知识体系而言确实是庞大的，系统学习它至少要以扎实掌握数学和计算机本科 6～7 门课程为基础。但是这些课程已经让很多在校学生“备受折磨”，更别提终日忙于工作与家庭、已毕业多年的企业管理与工程人员了。

如何让对数学久疏战阵或者本就不擅长数学的人快速领略和掌握人工智能与机器学习的全貌呢？

本书试图用通俗的语言讲解涵盖算法模型的机器学习。更进一步地，在深入浅出解析模型与算法之后，介绍使用 Python 相关工具进行开发的方法、解析经典案例，使读者做到**“能理解，能设计、能编码、能调试”**，真正将机器学习应用在自己的产品之中。

本书特色

1. 内容全面

全面覆盖了机器学习的三大领域：有监督学习、无监督学习、强化学习。在分析它们的传统算法模型后，着重解析近年来取得突破的深度学习在人工智能方面的应用。

2. 深入浅出

用生活化的语言描述算法与模型的原理与作用，并给出实践指导和案例解析。使得没有任何专业基础的读者在学习本书后能够独立设计与开发机器学习产品。

3. 工具多样

理论内容全面，以至于没有哪个工具能够全部实现这些模型，因此在每一个模型的实践部分选取最合适的工具。总体来看，本书围绕 scikit-learn 与 TensorFlow 展开实践，并在需要时引入其他工具。

4. 案例丰富

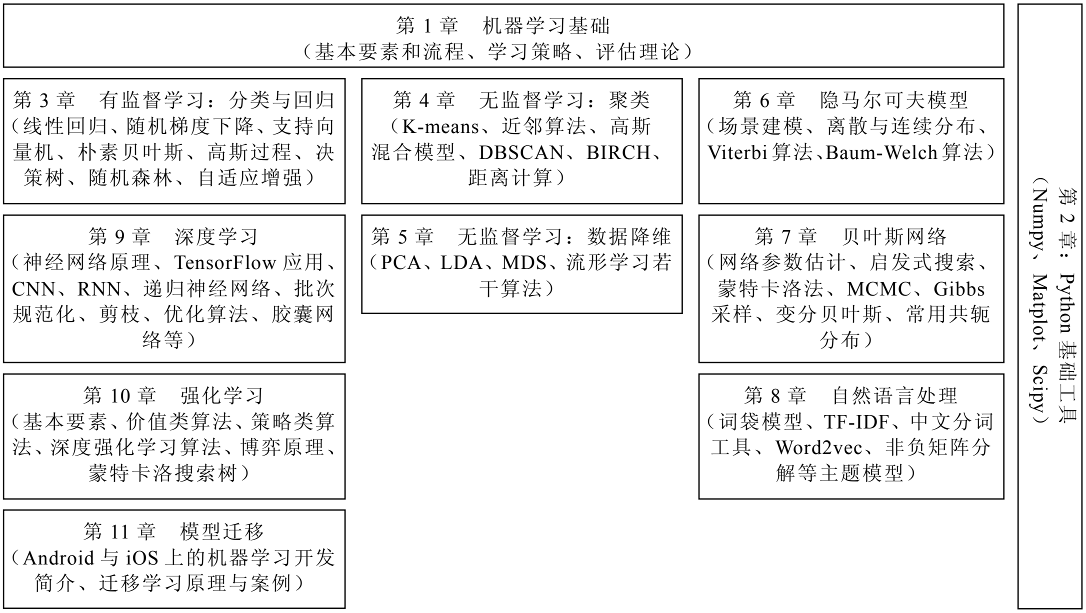
除了每个模型的小型实践，本书包括的较大案例是：金融预测、医疗诊断概率模型、月球登陆器、图像识别、写诗机器人、中国象棋博弈。

5. 授人以渔

在运用到较深的理论知识或更细节的论证结果时，本书给出这些知识与结论的出处，确保读者能够追本溯源；在工具使用方面，不仅着眼于接口细节，更关注那些能使读者快速形成查阅该工具最新在线文档的核心知识的能力。

本书内容体系

虽然本书每章有明确的主题内容，但知识体系有轻微依赖关系，因此对于普通读者来说建议按编排顺序阅读。对于有特定需求的读者，可以按照下图寻找学习路径。



除了第 2 章，学习上图中任何一个模块前，都建议以其上方的章节为基础。第 2 章介绍的是后续其他章节里实践所依赖的 Python 基础工具，对于只关注模型场景与原理的读者可将其跳过。

从图中可以看到，除了基础的第 1、2 章，本书由 3 条学习线组成。

（1）第 3、9、10、11 章

第 3 章以 scikit-learn 为工具介绍有监督学习的传统模型；第 9 章是本书篇幅最大的一章，以 TensroFlow 为工具学习近年来发展最快的深度学习模型，因为其与第 8 章有少量关联，所以放在了第 9 章；第 10 章学习强化学习传统算法与深度学习算法；第 11 章简单介绍深度学习模型的迁移方法。

（2）第 4、5 章

用 scikit-learn 讲解无监督学习模型原理、算法与应用。无监督学习中最主要的两部分

是聚类与降维，在讲解过程中比较了每个模型的优势与劣势。

（3）第 6、7、8 章

讲解概率类机器学习模型的原理与实战。第 6 章作为入门使读者领略概率模型的特点；第 7 章全面介绍贝叶斯类算法的基本知识和简单实践；第 8 章的重点是 LDA 主题模型，以第 7 章的内容为基础。

只要按照图中的顺序学习，几乎不需要任何基础就可以掌握图中所有机器学习领域的基本方法，为今后在这方面进行研究与工作打下坚实的基础。

本书读者

本书几乎适合任何对机器学习感兴趣的读者，比较典型的是：

◎ 产品设计经理。

◎ 技术管理者。

◎ 信息技术创业者。

◎ 数据分析员。

◎ 软件开发人员。

◎ 在校学生。

另外，如果您已经是机器学习领域的资深研究者，能看懂相关论文，或正准备在专业杂志上发表这方面的论文，可能这本书会不那么适合你。谨此说明。

感谢

感谢您的信任，如果阅读这本书能启发您获得新的灵感，那是我最大的荣幸。同时也因为本人水平有限，书中内容有疏漏之处也请赐教和包涵。

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

◎ **下载资源**：本书如提供示例代码及资源文件，均可在 下载资源 处下载。

◎ **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

◎ **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/35518

目　录

第 1 章　机器学习基础 ······ 1

1.1　引言 ······ 1

1.1.1　为什么使用机器学习 ······ 2

1.1.2　机器学习与数据挖掘 ······ 4

1.1.3　机器学习与人工智能 ······ 5

1.2　机器学习的一般流程 ······ 7

1.2.1　定义问题 ······ 7

1.2.2　收集数据 ······ 8

1.2.3　比较算法与模型 ······ 9

1.2.4　应用模型 ······ 10

1.3　学习策略 ······ 10

1.3.1　有监督学习 ······ 11

1.3.2　无监督学习 ······ 14

1.3.3　强化学习 ······ 16

1.3.4　综合模型与工具 ······ 18

1.4　评估理论 ······ 19

1.4.1　划分数据集 ······ 19

1.4.2　交叉验证 ······ 21

1.4.3　评估指标 ······ 22

1.4.4　拟合不足与过度拟合 ······ 25
1.5　本章内容回顾 ······ 26

第 2 章　Python 基础工具 ······ 27

2.1　Numpy ······ 28
2.1.1　Numpy 与 Scipy 的分工 ······ 28
2.1.2　ndarray 构造 ······ 29
2.1.3　数据类型 ······ 32
2.1.4　访问与修改 ······ 33
2.1.5　轴 ······ 35
2.1.6　维度操作 ······ 38
2.1.7　合并与拆分 ······ 40
2.1.8　增与删 ······ 41
2.1.9　全函数 ······ 42
2.1.10　广播 ······ 42
2.2　Matplot ······ 43
2.2.1　点线图 ······ 44
2.2.2　子视图 ······ 50
2.2.3　图像 ······ 53
2.2.4　等值图 ······ 57
2.2.5　三维绘图 ······ 58
2.2.6　从官网学习 ······ 59
2.3　Scipy ······ 60
2.3.1　数学与物理常数 ······ 61
2.3.2　特殊函数库 ······ 62
2.3.3　积分 ······ 64
2.3.4　优化 ······ 65
2.3.5　插值 ······ 67
2.3.6　离散傅里叶 ······ 68
2.3.7　卷积 ······ 70
2.3.8　线性分析 ······ 71
2.3.9　概率统计 ······ 73

2.4 本章内容回顾 ······ 77

第 3 章 有监督学习：分类与回归 ······ 79

3.1 线性回归 ······ 80
3.1.1 何谓线性模型 ······ 80
3.1.2 最小二乘法 ······ 81
3.1.3 最小二乘法的不足 ······ 82
3.1.4 岭回归 ······ 85
3.1.5 Lasso 回归 ······ 87
3.2 梯度下降 ······ 90
3.2.1 假设函数与损失函数 ······ 90
3.2.2 随机梯度下降 ······ 92
3.2.3 实战：SGDRegressor 和 SGDClassifier ······ 93
3.2.4 增量学习 ······ 94
3.3 支持向量机 ······ 95
3.3.1 最优超平面 ······ 95
3.3.2 软间隔 ······ 97
3.3.3 线性不可分问题 ······ 98
3.3.4 核函数 ······ 99
3.3.5 实战：scikit-learn 中的 SVM ······ 100
3.4 朴素贝叶斯分类 ······ 101
3.4.1 基础概率 ······ 102
3.4.2 贝叶斯分类原理 ······ 103
3.4.3 高斯朴素贝叶斯 ······ 105
3.4.4 多项式朴素贝叶斯 ······ 106
3.4.5 伯努利朴素贝叶斯 ······ 107
3.5 高斯过程 ······ 107
3.5.1 随机过程 ······ 108
3.5.2 无限维高斯分布 ······ 109
3.5.3 实战：gaussian_process 工具包 ······ 111
3.6 决策树 ······ 114
3.6.1 最易于理解的模型 ······ 114

3.6.2　熵的作用 …… 115
3.6.3　实战：DecisionTreeClassifier 与 DecisionTreeRegressor …… 117
3.6.4　树的可视化 …… 118
3.7　集成学习 …… 119
3.7.1　偏差与方差 …… 120
3.7.2　随机森林 …… 121
3.7.3　自适应增强 …… 124
3.8　综合话题 …… 126
3.8.1　参数与非参数学习 …… 127
3.8.2　One-Vs-All 与 One-Vs-One …… 127
3.8.3　评估工具 …… 129
3.8.4　超参数调试 …… 131
3.8.5　多路输出 …… 134
3.9　本章内容回顾 …… 134
第 4 章　无监督学习：聚类 …… 136
4.1　动机 …… 137
4.2　K-means …… 138
4.2.1　算法 …… 139
4.2.2　实战：scikit-learn 聚类调用 …… 141
4.2.3　如何选择 *K* 值 …… 144
4.3　近邻算法 …… 145
4.3.1　生活化的理解 …… 145
4.3.2　有趣的迭代 …… 146
4.3.3　实战：AffinityPropagation 类 …… 147
4.4　高斯混合模型 …… 149
4.4.1　中心极限定理 …… 150
4.4.2　最大似然估计 …… 151
4.4.3　几种协方差矩阵类型 …… 152
4.4.4　实战：GaussianMixture 类 …… 154
4.5　密度聚类 …… 156
4.5.1　凸数据集 …… 157

4.5.2 密度算法 158
4.5.3 实战：DBSCAN 类 159
4.6 BIRCH 160
4.6.1 层次模型综述 161
4.6.2 聚类特征树 162
4.6.3 实战：BIRCH 相关调用 164
4.7 距离计算 166
4.7.1 闵氏距离 166
4.7.2 马氏距离 167
4.7.3 余弦相似度 168
4.7.4 时间序列比较 169
4.7.5 杰卡德相似度 169
4.8 聚类评估 170
4.9 本章内容回顾 172
第 5 章 无监督学习：数据降维 173
5.1 主成分分析 174
5.1.1 寻找方差最大维度 174
5.1.2 用 PCA 降维 177
5.1.3 实战：用 PCA 寻找主成分 178
5.2 线性判别分析 181
5.2.1 双重标准 181
5.2.2 实战：使用 LinearDiscriminantAnalysis 183
5.3 多维标度法 185
5.3.1 保留距离信息的线性变换 185
5.3.2 MDS 的重要变形 187
5.3.3 实战：使用 MDS 类 188
5.4 流形学习之 Isomap 189
5.4.1 什么是流形 190
5.4.2 测地线距离 192
5.4.3 实战：使用 Isomap 类 193
5.5 流形学习之局部嵌入 195

5.5.1 局部线性嵌入 195
5.5.2 拉普拉斯特征映射（LE） 198
5.5.3 调用介绍 200
5.5.4 谱聚类 201
5.6 流形学习之 t-SNE 203
5.6.1 用 Kullback-Leiber 衡量分布相似度 203
5.6.2 为什么是 t-分布 205
5.6.3 实战：使用 TSNE 类 206
5.7 实战：降维模型之比较 207
5.8 本章内容回顾 210
第 6 章 隐马尔可夫模型 212
6.1 场景建模 213
6.1.1 两种状态链 213
6.1.2 两种概率 215
6.1.3 三种问题 217
6.1.4 hmmLearn 介绍 218
6.2 离散型分布算法与应用 222
6.2.1 前向算法与后向算法 222
6.2.2 MultinomialNB 求估计问题 226
6.2.3 Viterbi 算法 227
6.2.4 MultinomialNB 求解码问题 229
6.2.5 EM 算法 232
6.2.6 Baum-Welch 算法 233
6.2.7 用 hmmLearn 训练数据 235
6.3 连续型概率分布 236
6.3.1 多元高斯分布 237
6.3.2 GaussianHMM 239
6.3.3 GMMHMM 240
6.4 实战：股票预测模型 241
6.4.1 数据模型 241
6.4.2 目标 243

6.4.3 训练模型 ··· 243
6.4.4 分析模型参数 ··· 245
6.4.5 可视化短线预测 ··· 247
6.5 本章内容回顾 ··· 250

第 7 章 贝叶斯网络 ··· 251

7.1 什么是贝叶斯网络 ··· 252
7.1.1 典型贝叶斯问题 ··· 252
7.1.2 静态结构 ··· 253
7.1.3 联合/边缘/条件概率换算 ··· 256
7.1.4 链式法则与变量消元 ··· 258
7.2 网络构建 ··· 259
7.2.1 网络参数估计 ··· 260
7.2.2 启发式搜索 ··· 261
7.2.3 Chow-Liu Tree 算法 ··· 262
7.3 近似推理 ··· 263
7.3.1 蒙特卡洛方法 ··· 264
7.3.2 马尔可夫链收敛定理 ··· 265
7.3.3 MCMC 推理框架 ··· 267
7.3.4 Gibbs 采样 ··· 268
7.3.5 变分贝叶斯 ··· 268
7.4 利用共轭建模 ··· 270
7.4.1 共轭分布 ··· 270
7.4.2 隐含变量与显式变量 ··· 272
7.5 实战：胸科疾病诊断 ··· 274
7.5.1 诊断需求 ··· 274
7.5.2 Python 概率工具包 ··· 275
7.5.3 建立模型 ··· 276
7.5.4 MCMC 采样分析 ··· 278
7.5.5 近似推理 ··· 281
7.6 本章内容回顾 ··· 282

第 8 章　自然语言处理 ······ 284
8.1　文本建模 ······ 285
8.1.1　聊天机器人原理 ······ 285
8.1.2　词袋模型 ······ 286
8.1.3　访问新闻资源库 ······ 287
8.1.4　TF-IDF ······ 290
8.1.5　实战：关键词推举 ······ 290
8.2　词汇处理 ······ 294
8.2.1　中文分词 ······ 294
8.2.2　Word2vec ······ 296
8.2.3　实战：寻找近似词 ······ 298
8.3　主题模型 ······ 303
8.3.1　三层模型 ······ 303
8.3.2　非负矩阵分解 ······ 304
8.3.3　潜在语意分析 ······ 305
8.3.4　隐含狄利克雷分配 ······ 307
8.3.5　实战：使用工具包 ······ 309
8.4　实战：用 LDA 分析新闻库 ······ 311
8.4.1　文本预处理 ······ 311
8.4.2　训练与显示 ······ 313
8.4.3　困惑度调参 ······ 315
8.5　本章内容回顾 ······ 317
第 9 章　深度学习 ······ 319
9.1　神经网络基础 ······ 320
9.1.1　人工神经网络 ······ 320
9.1.2　神经元与激活函数 ······ 321
9.1.3　反向传播 ······ 323
9.1.4　万能网络 ······ 325
9.2　TensorFlow 核心应用 ······ 328
9.2.1　张量 ······ 329
9.2.2　开发架构 ······ 331

9.2.3 数据管理 …… 332
9.2.4 评估器 …… 335
9.2.5 图与会话 …… 338
9.2.6 逐代（epoch）训练 …… 341
9.2.7 图与统计可视化 …… 343
9.3 卷积神经网络 …… 349
9.3.1 给深度学习一个理由 …… 349
9.3.2 CNN 结构发展 …… 351
9.3.3 卷积层 …… 354
9.3.4 池化层 …… 356
9.3.5 ReLU 与 Softmax …… 357
9.3.6 Inception 与 ResNet …… 359
9.4 优化 …… 362
9.4.1 批次规范化 …… 362
9.4.2 剪枝 …… 364
9.4.3 算法选择 …… 366
9.5 循环神经网络与递归神经网络 …… 367
9.5.1 循环神经网络 …… 368
9.5.2 长短期记忆（LSTM） …… 371
9.5.3 递归神经网络 …… 374
9.6 前沿精选 …… 377
9.6.1 物件检测模型 …… 377
9.6.2 密连卷积网络 …… 381
9.6.3 胶囊网络 …… 382
9.7 CNN 实战：图像识别 …… 385
9.7.1 开源图像库 CIFAR …… 385
9.7.2 项目介绍 …… 388
9.7.3 构建 Graph …… 389
9.7.4 优化与训练 …… 392
9.7.5 运行 …… 394
9.8 RNN 实战：写诗机器人 …… 397
9.8.1 语言模型 …… 397

9.8.2　LSTM 开发步骤 1：网络架构 ······ 401
9.8.3　LSTM 开发步骤 2：数据加载 ······ 402
9.8.4　LSTM 开发步骤 3：搭建 TensorFlow Graph ······ 403
9.8.5　LSTM 开发步骤 4：解析 LSTM RNN ······ 404
9.8.6　LSTM 开发步骤 5：LSTM 中的参数 ······ 406
9.8.7　LSTM 开发步骤 6：用 sequence_loss 计算 RNN 损失值 ······ 406
9.8.8　LSTM 开发步骤 7：学习速度可调优化器 ······ 407
9.8.9　LSTM 开发步骤 8：训练 ······ 408
9.8.10　开始写唐诗 ······ 410
9.8.11　写唐诗步骤 1：用唐诗语料训练语言模型 ······ 410
9.8.12　写唐诗步骤 2：作诗 ······ 412
9.8.13　写唐诗步骤 3：作品举例 ······ 414
9.9　本章内容回顾 ······ 415

第 10 章　强化学习 ······ 418

10.1　场景与原理 ······ 419
10.1.1　借 AlphaGo 谈人工智能 ······ 419
10.1.2　基于价值的算法 Q-Learning 与 Sarsa ······ 421
10.1.3　基于策略的算法 ······ 424
10.1.4　基于模型的算法 ······ 426
10.2　OpenAI Gym ······ 427
10.2.1　环境调用 ······ 428
10.2.2　实战：用 Q-Learning 开发走迷宫机器人 ······ 432
10.3　深度强化学习 ······ 435
10.3.1　DQN 及改进 ······ 435
10.3.2　DPN、DDPG 及 A3C ······ 436
10.3.3　实战：用 DPN 训练月球定点登陆 ······ 439
10.4　博弈原理 ······ 444
10.4.1　深度搜索与广度搜索 ······ 444
10.4.2　完美决策 ······ 446
10.4.3　蒙特卡洛搜索树 ······ 448
10.5　实战：中国象棋版 AlphaGo Zero ······ 449

10.5.1 开源版本 AlphaGo Zero …… 450
10.5.2 盘面建模 …… 452
10.5.3 左右互搏 …… 457
10.5.4 MCTS 详解 …… 464
10.5.5 DDPG 详解 …… 468
10.5.6 运行展示：训练 …… 473
10.5.7 运行展示：查看统计 …… 475
10.5.8 运行展示：当头炮、把马跳 …… 475
10.5.9 运行展示：人机博弈 …… 476
10.6 本章内容回顾 …… 477

第 11 章 模型迁移 …… 478

11.1 走向移动端 …… 478
11.1.1 Android 上的 TensorFlow …… 479
11.1.2 iOS 上的 CoreML …… 480
11.2 迁移学习 …… 483
11.2.1 动机 …… 483
11.2.2 训练流程 …… 484
11.3 案例实战：基于 TensorFlow Hub 的迁移学习开发 …… 485
11.3.1 下载并训练 …… 485
11.3.2 检验学习成果 …… 486
11.3.3 迁移学习开发 …… 487
11.4 本章内容回顾 …… 488

后记 …… 489

1

第 1 章 机器学习基础

思考：云计算和大数据，与机器学习的关系是什么？

这是一个最好的时代——对于机器学习来说。云计算和大数据是近十年来信息技术发展的最大主题，但它们绝不是直接改变人类生活方式、提供商业模式的技术，它们的发展和成熟给机器学习、人工智能等技术的应用打下了坚实的基础。

1.1 引言

当下有越来越多的消费类产品被冠以“智能”二字，以智能为主题的科幻电影也比以前拥有更多的观众，这些使得人们越来越多地思考：智能设备到底学到了什么？它们如何理解人类指令？它们会背叛人类吗？

同时，机器学习和人工智能已经成为科技企业、技术专家和管理者最感兴趣的话题，

把机器学习和人工智能技术引入产品中并给终端用户提供新的服务，是他们的战略目标。

1.1.1　为什么使用机器学习

在 20 世纪末电视节目不算丰富的年代，美国动画片《变形金刚》是当时很多孩子每天晚上最渴望的节目。大多数孩子当时都有自己偏爱的角色，擎天柱、大黄蜂、机器恐龙、甚至是惊破天，而令我印象最深刻的却是一个出场不多的组合机器人——“计算王”。它的厉害之处是，在任何战斗前都能根据当时的形势计算出打败对手的可能性，从而可以在胜算不足时开溜，做到只打有把握之仗，这使它能在大多数情况下立于不败之地。

不论变形金刚在创作时是否聘请了人工智能方面的专家当顾问，现在回想起来“计算王”这个强大角色体现的正是机器学习中的有监督学习技术的应用。如果读者现在还不太能区分人工智能、机器学习、有监督学习这些概念的区别，那么本节是一个最好的开始。

思考：人工智能、机器学习、有监督学习、无监督学习有什么区别？

1. 人工智能问题的复杂性

自计算机科学问世以来，人们就期望计算机有足够的知识来帮人类解决各种各样的问题。但曾经在很长一段时间里，计算机能解决的只是数字的加减乘除、文字存储传输、固定知识的问答等固定模式的问题。

在机器学习技术之前的人工智能科学家倾向于将所有的人类基本规则告诉计算机，以期在遇到复杂问题时计算机能够自动进行知识推理。这样的模式可以让计算机精确地完成加减乘除这样的简单任务，但当局面越来越复杂时，计算能力却不堪重负。一方面是因为总结这些基本规则的困难性，另一方面，随着规则数量的增加，对推理计算能力的要求成指数级增长。这导致了 20 世纪 80 年代人工智能发展的停滞。

以自然语言理解为例。在基于语法规则的系统中，语言专家需要将经过整理的各种语言的单词、词性、基础语法告诉计算机，然后计算机基于这些规则对给定的语句建立语法树以理解句子的含义。语法树举例如图 1-1 所示。

该图取自于《Python 自然语言处理》，其中的英文短语符号表示短语在句子中的成分类型，比如 VP（Verb Phrase）是动词短语，NP（Noun Phrase）是名词短语。计算机通过规则推理由“同时就带了四张熟牛皮和十二头肥牛”这段文字产生如图 1-1 所示的语法树的复杂度大得惊人。

提示：“同时就带了四张熟牛皮和十二头肥牛”这句话来源于战国时期“弦高救国”的故事，故事中郑商弦高用这些礼品犒劳秦军，避免了秦军侵略郑国。

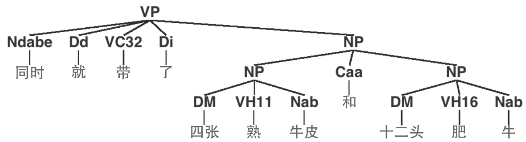


图 1-1　语法树举例

这种系统的挑战性在于宽广的文字空间和无处不在的歧义。随着句子长度的增加，复杂度以无法想象的速度增长。分析树的解析数量是随着句子长度成 Catalan 数增长的，当句子长度为 50 个字时，需要建立 10^{12} 个语法树以找到最佳匹配。

提示：Catalan 数是以比利时数学家欧仁·查理·卡塔兰命名的一种统计用数列。其公式是 $C_n = \dfrac{(2n)!}{(n+1)!n!}$，比如当 n=1、2、3、4、5、6、7、8、9…时数列是 1、2、5、14、42、132、429、1430、16796…，数列越向后增长越快。

2. 机器学习的定义

科学家不断地探索能使计算机自己学习基本规则并整理规则之间的联系的方法，成为现代人工智能技术的重要组成部分。在全球使用最广泛的机器学习教材——卡内基梅隆大学的《机器学习》一书中，对完成这类工作的机器学习给出了定义：

如果一个计算机程序针对某类任务 T 可以用性能 P 衡量，并且能通过经验 E 来自我完善，则计算机可以在经验 E 中学习任务 T，这就是机器学习方法。

在这个定义中明确提出了每一个机器学习系统必备的三个元素。

◎ 任务：有明确的目标，这些目标可以是打败一流围棋选手的计算机程序、识别阿拉伯语的手写字符、找出星巴克咖啡馆的最佳选址地点等。但不能是诸如“让系统更智能”这样的模糊概念。

◎ 可衡量：对任务有定量的评价方法，系统通过它来改进性能。对于围棋程序，可以通过其击败人类选手的等级来衡量；手写识别可以用通用文本库识别率来衡量。

◎ 可成长：这在现代机器学习中称为适配、训练。围棋程序可以通过与自己下棋获取经验；手写识别可以根据语料库中大量的已识别信息提高自己。

思考：有哪些生活中常见的机器学习系统？

通过这样的学习系统，计算机已经能完成很多复杂的工作。比如用于全球旅游的自助语言翻译系统、医院里从医疗记录中找到最佳治疗方案的智能诊断系统、在一定程度上脱离驾驶员控制的自动驾驶系统、虚拟商店里的智能导购机器人等。这些领域都用机器学习技术取得了突破性的进展。

3. 与机器学习有关的其他学科

机器学习的发展以很多其他学科为基础，包括传统计算机、数据库与数据仓库、信息论、人工智能、计量经济学、统计学、神经科学等。它们之中的大多数是机器学习的理论与实践基础：

- ◎ 计算机提供基础的计算与存储能力。
- ◎ 数据库进行结构化数据的保存、查询、修改等；数据仓库 ETL 几乎是很多机器学习任务的必要环节。
- ◎ 信息论中的熵理论是很多人研究机器学习算法（如决策树、最大熵算法）的理论基础。
- ◎ 计量经济学中的回归被直接引用在机器学习中。
- ◎ 神经网络与深度学习的灵感来自于神经科学。

提示：数据仓库中的 ETL 是英文 Extract-Transform-Load 的缩写，即数据抽取、转换和加载。

另一方面，数据挖掘与人工智能是两个与机器学习相互影响的领域，也常有人误用这三个概念。

1.1.2 机器学习与数据挖掘

在 2001 年麻省理工学院出版的《数据挖掘原理》（*Principle of Data Mining*）一书中，提供了数据挖掘的一个非常简单的定义：

一门从大量资料或者资料库中提取有用信息的科学。

可以看出，数据挖掘强调的只是一个“提取有用信息”的目标，并没有像机器学习那样定义了方法或手段。而随着后来的发展，数据挖掘与机器学习采用了越来越多相同的方

法，比如分类、回归、聚类等都是两个学科的共同目标任务。

在不同点方面，机器学习学到的知识通常是一个普适或可以被广泛应用的知识，比如手写识别、自动驾驶。这些知识一旦被掌握，可以迅速普及。而数据挖掘常常是针对某个特定的项目或数据集，被挖掘的知识更适用于特定的服务对象，比如挖掘某个超市中最值得销售的商品。由于每个超市所在社区与居民文化的不同，往往需要根据每个超市自身的销售历史数据进行各自挖掘。

注意：最值得销售的商品并不单单是利润最高的商品，需要结合利润、周转率、仓储体积、过期风险等进行综合衡量。

如图 1-2 所示从目标、手段、场合等不同方面演示了机器学习与数据挖掘的主要异同点。可以得知两者在方法与算法方面是互通互用的，是两门学科在各自领域最主要的研究课题，这些也是本书后续章节的主要内容。它们的不同之处主要在于出发点的不同：数据挖掘更强调流程、强调结果，而机器学习强调对算法本身的研究。

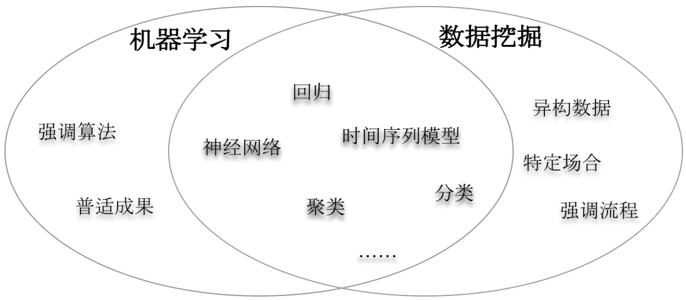


图 1-2　机器学习与数据挖掘的主要异同点

可以肯定的是，一个机器学习专家只需花很少的时间就能成为一个数据挖掘专家，反之应该也是如此。

思考：机器学习与数据挖掘的异同还有哪些？

1.1.3　机器学习与人工智能

人工智能（Artificial Intelligence，AI）是当下最热门的科技词汇，但很多人其实不知道当他们在说“人工智能”时实际是在说机器学习。

人工智能最被认可的定义来自阿兰·图灵于 1950 年提出的图灵测试验证法：

如果一台计算机能用书面方式回答人类提出的问题，并且一位人类询问者在收到回答后意识不到这是来自于计算机的回答，那么这台计算机就拥有了人工智能。

显然，现在市场上的“智能”产品几乎都无法通过图灵测试。

在被全世界多数大学采用的人工智能教材《人工智能——一种现代的方法》中认为能通过图灵测试的计算机应有如下 4 个方面的能力。

◎ 自然语言处理：使其能成功用汉语、英语等与人类交流，而不是用固定的表格、关键字提示等方式。

◎ 知识表示：存储和传输计算机所了解的知识。这可以看成一种中间语言，其足够强大以表征所有类型的知识，包括名词、逻辑、运算等。

◎ 自动推理：计算机无法存储所有知识，那么用已有知识解决新的问题就是一种必需的知识。最简单的推理就是三段论：金属可以导电（大前提）→铜是金属（小前提）→铜可以导电（结论）。

◎ 机器学习：适应新的情况并用适配模式预测和解决问题。

可知，机器学习是人工智能的一部分，现在诸如导航软件、语音翻译等其实都是一种机器学习产品，如图 1-3 所示是机器学习与人工智能的关系。

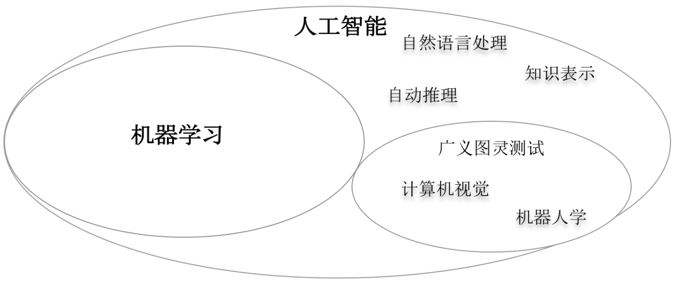


图 1-3　机器学习与人工智能的关系

注意：图 1-3 中的广义图灵测试是指为人工智能加入物理特性的测试。

由于近代机器学习方法在借鉴统计理论后得到了长足发展，它越来越多地影响到了人工智能的其他方面。比如在自然语言处理领域，当前很多网络店铺的虚拟客服能在很大程度上解决一些客户用自然语言提出的售后问题，其背后正是采用了基于机器学习方法的客户意图分类和搜索系统。

1.2　机器学习的一般流程

虽然机器学习科学包含了大量解决不同问题的算法与技术，但在工程实践中它还是有一个几乎普适的流程模板，如图 1-4 所示。

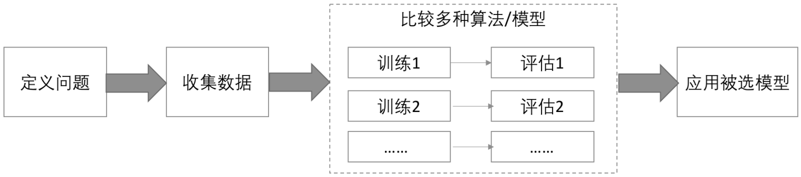


图 1-4　机器学习项目流程

1.2.1　定义问题

在机器学习的定义中有提到所有的机器学习项目必须有一个明确的任务目标，所以这是要解决的第一个问题。万事开头难，虽然这一步骤不需要具体的算法或人工智能技术，但却需要提出者具有敏锐的眼光和全局意识，而这些通常以丰富的系统知识为基础，是机器学习项目能否成功非常关键的一步。正因如此，大多数科技企业都有类似“创意基金”这样激励员工定义、提出问题的措施。

以下几个步骤可以帮助读者正确定义机器学习的问题：

◎　明确动机。

◎　列出已有的条件。

◎　描绘解决思路。

假设有一个胸科医院的自动诊断项目，可以通过如下步骤将对问题的定义具体化。

◎　动机：由于专家级医生资源紧缺，同时为了减少医生疲劳产生的误诊，需要开发一套自动为病人诊断疾病的系统。

◎　已有条件：医疗研究机构积累了大量各种病人的病理特征。对于来医院诊断的病人，可以通过胸片、验血等检验手段获得病人较全面的病理特征。

◎　解决思路：用历史病人特征建立疾病适配模型，用“疾病模型+病人病理特征”预测病人可能的疾病。

思考：假设有一个医院导航智能机器人项目，又该如何定义问题呢？

1.2.2　收集数据

现在可以开始收集和整理建立模型所需要的数据了。数据采集与清理工作比较琐碎，虽然不是机器学习学科研究的重点，但由于每个项目有自己的数据特点，缺乏统一的算法，经常是项目中需要花费大量人力与时间的阶段。一般通过数据采集、数据预处理、数据转换三个步骤完成本阶段。

1. 数据采集（Data Selection）

需要根据数据的不同形式和所在物理媒介的不同而展开。从数据形式上看，数据可能是文本、数字、声音、图像等。从物理媒介上看，数据可能来自机构内部、互联网或实时采集等。当前已经有诸多开源的或商业机构提供的不同数据源的便捷访问软件包，如图 1-5 所示为商业智能软件公司“微策略”提供的常用数据源。



图 1-5　商业智能软件公司“微策略”提供的常用数据源

本步骤的目的除了要使学习系统能够快速读取不同数据源的信息，还定义了这些数据源中的哪些数据是学习所需、哪些数据需要从学习系统中排除。对于胸科诊断来说，病人职业、血样指标、胸片诊断等都是必要信息；而像病人星座、政治面貌等一般可认为与疾病无关，可以从学习系统中排除。

提示：有些读者可能认为无用数据加入分析也无妨，其实在回归统计理论中，无用变量会影响方差估计，其他学习模型中也会由于数据的有限性使无效数据成为一种噪声。

2. 数据预处理（Data Preprocessing）

所谓的数据预处理是机器学习和数据挖掘中的必要步骤，通常包含数据清理、格式化等。对于胸科诊断，包括以下内容。

- ◎ 清理：排除特征不完全的案例。比如某些历史病例丢失了病人体重、血样等重要信息，无法成为有效案例。
- ◎ 格式化：机器学习模型需要以数字作为输入，所以需要将其他形式的信息转换为数字，有时还可以在这里先进行排序处理。比如在本例中，将胸片中的有无骨折、气腹等诊断转换为布尔/数字特征。

3. 数据转换（Data Transformation）

在这里需要对已有的数据进行编辑、合并或拆分，使其可以直接作为机器学习模型的输入。比如在胸科诊断中，用病人的身高+体重合成体重指数表示肥胖程度，将生日与发病日期合成得到发病年龄指标等。

1.2.3　比较算法与模型

因为机器学习是对未知知识的探索，通常在本阶段需要探索不同模型、调试各种超参数以获得最优模型。而衡量不同模型的优劣需要有一个客观标准，这就是所谓的评估技术。

本阶段是真正的分析与学习阶段，其目的是利用已经准备好的数据进行已定义问题的学习。这里先明确几个需要在机器学习领域反复用到的概念。

- ◎ 模型（Model）：在机器学习中用来表征已知的历史数据集。每种算法、参数会产生不同的模型。模型被直接用于学习后的预测和分类应用。
- ◎ 训练（Training）：也称适配（Fitting），是指根据数据集计算出模型的过程。
- ◎ 预测（Predict）：用新的案例特征在模型上应用，以产生对新数据的判断。

◎　评估（Evaluation）：检验模型效果的手段，本章最后一节会进行更详细的描述。

如图 1-6 所示是模型、训练、预测之间的关系。其中的关键点是，机器学习在预测时使用的是模型，而不是直接使用历史数据。这是区别于传统的查找和人工智能算法的根本所在。本书的主要内容就是描述基于不同算法的模型原理与实践。

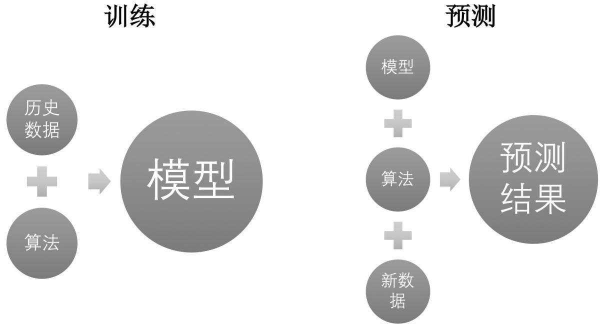


图 1-6　模型、训练、预测之间的关系

在反复训练时，经常需要调节所谓超参数（Hyperparameter），它是在机器学习模型训练之前人为设定的参数，通常需要根据项目和算法经验进行调整。本书后续章节将对每个算法所涉及的重要超参数进行解析。

1.2.4　应用模型

数据收集与生成模型工作通常需要大量的存储与计算资源，同时现代机器趋势也越来越倾向于分布式的预测能力。通常在完成第三阶段的训练与模型选择后，需要将模型部署到正式的生产环境中或迁移到移动设备上。

与收集数据阶段类似，本阶段也是在不同学习领域差别较大的阶段，但不是本书的主要内容，只在最后一章略作探讨。

1.3　学习策略

经过数十年的发展，机器学习已先后衍生出百余种算法，几乎每种算法又有若干分支。这些算法根据动机和适用场景可分为三大类型：有监督学习、无监督学习、强化学习。

如图 1-7 所示是机器学习分类图。

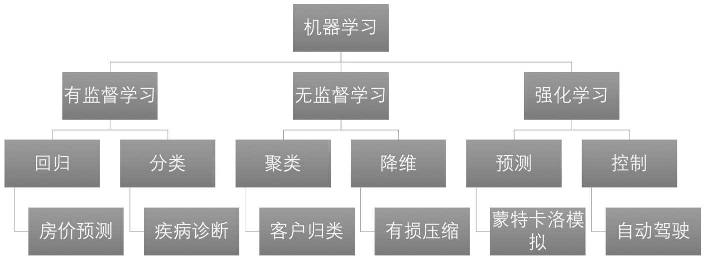


图 1-7　机器学习分类图

在图 1-7 中第三行是对学习策略的进一步分类，第四行是典型应用场景举例。除了这三大经典策略，还有一些模型综合了有监督学习和无监督学习的特点，本书不再细分。

1.3.1　有监督学习

有监督学习（Supervised Learning）是指这样的一种场景：

有一组数量较多的历史样本集，其中每个样本有一组特征（Feature）和一个或几个表示其自身的类型或数值的标签（Label）；对历史样本进行学习得到模型后，可以用新样本的特征预测其对应的标签。

1. 场景

在有监督学习中，可以将每条数据看成一条由特征到标签的映射，训练的目的是找出映射的规律。根据标签的类型可以将有监督学习再分为两个子类。

◎　分类（Classification）：标签是可数的离散类型，比如疾病诊断（疾病的类型有限）、图像文字识别（文字的总量有限）。

◎　回归（Regression）：标签是不可数的连续类型，有大小关系，比如房价预测（值无法枚举）。

如图 1-8 所示是一个胸科诊断的分类案例。



图 1-8　胸科诊断的分类案例

图 1-8 中的年龄、血液 pH 值、是否吸烟就是模型的特征，诊断结果（肺气肿/正常）是学习的标签。

注意：图 1-8 中分类问题的特征变量也可以是连续类型的（年龄、pH 值）。

2. 算法

有监督学习是机器学习中最易理解、发展最成熟的一个领域，其应用最广泛，算法可以分成以下几类。

- ◎ 线性分析法（Linear Analysis）：来源于统计学，其中众所周知的最小二乘法（Ordinary Least Square，OLS）是优化目标最易理解的回归学习算法，通过对优化目标的调整还衍生了 Ridge Regression、Lasso Regression 等算法。此外还包括线性判别分析（Linear Discriminant Analysis）。
- ◎ 梯度下降法（Gradient Descent）：用于寻找函数最小值或最大值。主要包括 3 个分支，批量梯度下降法 BGD、随机梯度下降法 SGD、小批量梯度下降法 MBGD。
- ◎ 朴素贝叶斯（Naive Bayes）：基于概率论的分类方法。在贝叶斯理论中，该方法要求所有特征之间相互独立，但在 2004 年 Harry Zhang 的论文 *The Optimality of Naive Bayes* 中阐述了当特征之间有比较平和的关联时朴素贝叶斯也能达到很好的效果。
- ◎ 决策树（Decision Tree）：源自风险管理的辅助决策系统，是一个利用树状模型的决策支持工具，根据其建分支的策略不同派生了很多子算法，如 ID3、C4.5、CART 等。其优点是学习结果易于人类理解，缺点是当数据集变化时决策图变化较大。
- ◎ 支持向量机（Support Vector Machine，SVM）：20 世纪 60 年代就被提出，直到

1992 年由 Bernhard E.Boser 等人改进为可以应用于非线性问题后被广泛应用，在 21 世纪初期的很长时间里被认为是最好的分类器。

◎ 神经网络（Neutral Network，NN）：由名称可知源于生物神经学，具有较长历史，可以处理复杂的非线性问题。传统神经网络的研究曾一度停滞，但随着计算机计算能力的提升和卷积网络结构的提出，由其发展而来的深度学习（Deep Learning）已经成为当前机器学习中非常强大的工具。

◎ 集成学习（Ensemble Learning）：是一种利用若干个基础分类器共同执行决策的方法。此方法近来被广泛应用，其中的随机森林（Random Forrest）正在逐步取代 SVM 的地位。此外，还有以 AdaBoost 为代表的提升方法（Boosting Method）。

所有的有监督学习算法都有一定的容错性，即不要求所有历史样本绝对正确、可以有部分标签被错误分配的样本。当然，样本中的错误越多越不容易训练出正确的模型。

3. 回归与分类的关系

读者应该已经发现：虽然有监督学习的适用场景可以分成两类，但介绍算法时并没有区分哪些适用于回归，哪些适用于分类。其实大多数的算法都可以同时处理这两类问题。如图 1-9 所示，假设某算法可以处理回归问题，那么当然可以将其值域划分成可数的几段用以表征分类问题。

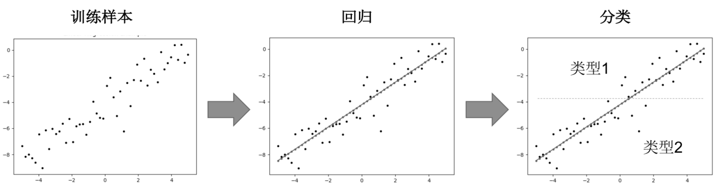


图 1-9　回归模型可以解决分类问题

图 1-9 中，左图是训练的原始样本；用线性回归学习后可以得到中图的回归线，回归线上的点就是之后的预测点；右图示意对回归结果设置阈值可以用来表达分类问题。从这个角度看，回归与分类的区别只不过是从不同角度分析学习结果而已。

另一方面，如果一个模型可以解决分类问题，则在分类类别之间做线性插值就是一种最简单的将分类模型转化为回归模型的方法。

因此，与很多教程书籍不同，本书不刻意区分它们，将有监督学习算法详细原理与实践的介绍统一放在了第 3 章。此外考虑内容的连贯性，笔者将神经网络方面的内容放在了

深度学习章节。

1.3.2 无监督学习

有监督学习用于解决分类问题的前提是，必须有一个带标签数据的样本集，但获得数据标签的代价往往是非常昂贵的。同时，这些标签通常都是人工标注，标注错误的情况也时有发生。这样就促使了无监督学习策略的发展，简单地说就是：

对无标签数据进行推理的机器学习方法。

1. 场景

由于无监督学习的前提是不需要前期的人类判断的，所以它一般作为某项学习任务的前置步骤，用于规约数据；在无监督学习之后，需要加入人类知识以使成果有实用价值。在人类知识加入的时间点比较两种学习策略，如图 1-10 所示。

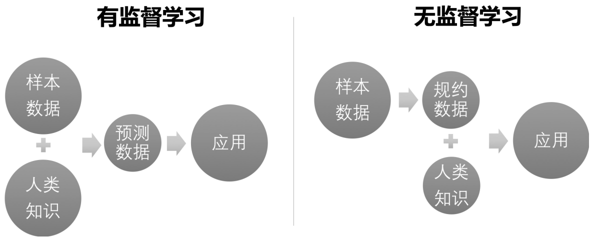


图 1-10　有监督学习与无监督学习的比较

一般来说，人类理解由无监督学习规约后的数据比整理样本数据中的标签更容易些，所以总体上无监督学习需要更少的人工参与。

无监督学习的算法比较丰富，按整理数据的方式有如下两大分支。

- ◎ 聚类（Clustering）：最主要的无监督学习方式，指将已有的样本数据分成若干个子集。生成的模型也可用于为新样本划分类别。
- ◎ 降维（Dimensionality Reduction）：即以保持数据之间现有距离关系不变为目标，将高维数据转换为低维数据。

此外还有一些小的算法族群，比如协方差分析（Covariance Estimation）、边缘检测（Outlier Detection）等。

如图 1-11 所示举例说明无监督学习最重要的方式——聚类的适用场景。它是一个银行客户的聚类示意图，将已有的客户总体分成两个子集。在进行聚类训练后，新客户也可以用已有的模型划分到相应子集。

总体

	年龄	存款额(万)	贷款额(万)	......
样本1	29	2	10	......
样本2	56	30	0	......
样本3	47	2390	1030	......
样本4	82	10	0	......
样本5	40	746	57	......
......	......	......	......	......

聚类

子集1

	年龄	存款额(万)	贷款额(万)	......
样本1	29	2	10	......
样本2	56	30	0	......
样本4	82	10	0	......
......	......	......	......	......

子集2

	年龄	存款额(万)	贷款额(万)	......
样本3	47	2390	1030	......
样本5	40	746	57	......
......	......	......	......	......

图 1-11　聚类的适用场景举例

聚类只是提供子集划分方案，而划分的逻辑意义需要人类进行辨别。在图 1-11 中，从结果看算法将所有客户按存款额和贷款额的多少分成了两类。对于大多数银行来说，子集 1 对应的是普通用户，子集 2 对应的是重要客户。

2. 聚类算法

聚类算法仍然是当下一个不断发展的领域，各种方法比较繁杂。本书主要学习目前比较成熟的几种聚类策略，如下所示。

◎ 距离切分方法（Partition Method）：一种最基础的算法，根据特征之间的距离进行聚类划分。具体算法主要是指 K-means 及其派生算法。

◎ 密度方法（Density Method）：通过定义每个子集的最小成员数量和成员之间距离实现划分。最典型的算法是 DBSCAN，即 Density-Based Spatial Clustering of Applications with Noise。

◎ 模型方法（Model Method）：以概率模型（以高斯混合模型为典型，即 Gaussian Mixture Model）和神经网络模型（Self Organizing Map，SOM）为主要代表。其特点是不完全将样本认定为属于某子集，而是指出样本属于各子集的可能性的

大小。

◎ 层次方法（Hierarchical Method）：不同于其他聚类将总体划分成彼此地位平等的多个子集，层次方法最终将数据集划分成有父子关系的树形结构。这样就可以在聚类的同时考察各子类之间的亲缘关系，比较典型的是 BIRCH（Balanced Iterative Reducing and Clustering using Hierarchies）模型。

3. 降维算法

如前所述，降维一般被用来压缩特征数量以便后续处理，其相对聚类来说略显抽象。本书介绍如下两类降维策略。

◎ 线性降维：顾名思义用来处理线性问题。模型比较简单，包括常见的主成分分析（Principle Component Analysis，PCA）和线性判别分析（Linear Discriminant Analysis，LDA）。

◎ 流行学习（Manifold Learning）：是近期学术界的热点，可以处理非线性降维。目前比较成熟的算法包括 Isomap、局部线性嵌入（Locally Linear Embedding，LLE）等。

本书第 4、5 章将分别详细讨论聚类和降维的主要算法原理与实践。

1.3.3 强化学习

强化学习是对英文 Reinforced Learning 的中文翻译，它的另一个中文名称是“增强学习”。相对于有监督学习和无监督学习而言，强化学习是一个相对独特的分支；前两者偏向于对数据的静态分析，后者倾向于在动态环境中寻找合理的行为决策。

强化学习的行为主体是一个在某种环境中独立运行的 Agent（可以理解为“机器人”），它可以通过训练获得在该环境中的最佳行为模式。强化学习被看成最接近人工智能的一个机器学习领域。

思考：为什么说强化学习是最接近人工智能的一个机器学习领域？

1. 5 个要素

强化学习的场景由如下两个对象构成。

◎　智能代理（Agent）：是可以采取一系列行动以达到某种目标的控制器，可以形象地将其理解为机器人大脑。比如自动驾驶的控制器、打败李世石的 AlphaGo。

◎　环境（Environment）：是 Agent 所能感知和控制的世界模型。对自动驾驶来说，Environment 就是 Agent 所能感知到的路况和车本身的行驶能力；对 AlphaGo 来说，Environment 包括棋盘上的每种状态和下棋规则。

这两个对象其实定义了机器人和其所能感知到的世界。就像人类能在自己的世界中行走、享受阳光，机器人也可以通过 3 种方式与其所在的环境进行交互。

◎　状态（State）：是任意一个静态时刻 Agent 能感知到的 Environment 情况，相当于某一时刻人类五官能感知到的一切。

◎　行为（Action）：是 Agent 能在 Environment 中执行的行为，对应人类四肢所能做的所有事。

◎　反馈（Reward）：是 Agent 执行某个/某些 Action 后获得的结果。Reward 可以是正向的或者是负向的，相当于人类感受到的酸甜苦辣。

以上 5 种强化学习要素的关系如图 1-12 所示，它们在一起构建了强化学习的应用场景。



图 1-12　强化学习场景

另外，在强化学习中 Reward 有时是延时获得的。即 Agent 在做出某个 Action 后不会马上获得 Reward，而需要在一系列 Action 之后才能获得。每个任务最终获得的 Reward 被称为 Value。比如在围棋环境中，只有结果是胜或败才对之前的所有 Action 给出最终的 Value。

延迟获得 Value 的本质分析的是一系列相关行为共同发生的作用，也是强化学习与有监督学习最主要的一个不同点。试想如果每一个 Action 都可以获得一个相应的 Reward，

那么 Reward 就退化成了有监督学习中的 label（标签）。

2. 两种场景与算法

具备上述五个要素的强化学习可以用来解决如下两类问题。

◎ 状态预测问题：用马尔可夫过程估计在任一时刻各种状态发生的可能性，其中蒙特卡洛模拟（Monte Carlo Method）是一类重要方法。

◎ 控制问题：如何控制 Agent 以获得最大 Reward。其算法可以分成如下两类。

i．基于策略的学习（Policy-Based）：基于概率分布学习行为的可能性，根据可能性选择执行的动作，可学习连续值或离散值类型行为。典型算法是 Policy Gradients。

ii．基于价值学习（Value-Based）：直接基于 Reward 学习行为结果，只能学习离散类型行为，包括算法 Q-Learning、Sarsa。

另外，还有个别算法兼具 Policy-Based 和 Value-Based 特点，比如 Actor-Critic。

不得不承认的是，虽然强化学习是更智能的机器学习分支，但目前产品级应用还比较少，多集中在游戏娱乐和简单工业控制领域。本书将在第 6 章介绍隐马尔可夫模型，第 7 章介绍以马尔可夫收敛定理为基础的蒙特卡洛推理，在第 10 章详细学习各类强化学习控制问题。

1.3.4 综合模型与工具

除了三大类型的学习策略，还有如下几个重要的综合性工具。

◎ 隐马尔可夫模型（Hidden Markov Model，HMM）：基于概率转换的双层状态链模型，该技术曾大幅度提高了语音识别、智能拼音输入等领域产品的准确率。

◎ 贝叶斯网络（Bayesian Network）：基于条件概率结点的网络模型，用贝叶斯理论推测被预测事件发生的概率。足球比赛前的胜负赔率预测、NBA 赛季前的总冠军预测等，这些都是贝叶斯网络的典型应用。

◎ 主题模型（Topic Model）：用三层结构分析文档的内涵主题，是文本分析的主要研究方向之一，逐步产生了 LSA、LDA、HDP 等综合性算法。可用于文档情感分析、相似文档归类等。

本书将在第 6、7、8 章分别介绍它们的原理与开发框架。

此外，深度学习（Deep Learning）是当前最热门的人工智能话题，它是一种用于解决机器学习问题的技术框架，可以用于解决所有三大类机器学习问题。谷歌于 2015 年 11 月发布了开源框架 TensorFlow，完备的生态链使得它在发布后迅速成为了最主要的深度学习软件开发框架。可以将传统机器学习模型与深度学习模型看成常规武器与核武器的关系，轻量级的传统模型适用于快速开发实践，重量级的深度学习适合精度更高的知识挖掘，本书第 9 章将从神经网络开始逐步扩展到深度学习的几个典型，以及在 TensorFlow 中的开发方法。

1.4　评估理论

笔者在 1.2 节中指出，在机器学习项目开发流程中需要尝试多种算法与参数，并在比较后选择最优模型部署，所以如何评估模型的优劣成为一个必要的步骤。本节介绍评估的常用术语和方法。

1.4.1　划分数据集

任何机器学习算法都是基于对已有数据集或环境的信息挖掘，要求将从现有数据学习得到的模型能够适配于未来的新数据。

1. 训练集（Training set）与测试集（Test set）

很自然的，在评估模型能力的时候需要采用与模型训练时不同的数据集，因此在训练模型之前需要将已有训练集与测试集划分成如图 1-13 所示的两部分。

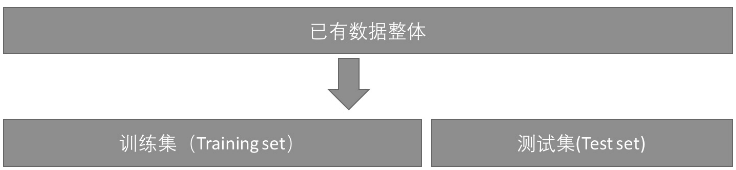


图 1-13　训练集与测试集的划分

顾名思义，图中的训练集用于在训练模型时使用，测试集用于评估模型准确率。一般

训练集与测试集一旦划分就无须再变动，因为只有稳定的测试集才能用来衡量不同模型的准确率。一旦重新划分两个集合，那么需要重新训练所有模型并在新的测试集上进行评估。

2. 随机采样（Random Sampling）

一般来说两个集合的划分需加入随机因子，使得每个数据项有相等的机会被分到任一集合中。如不加入随机因子，可能出现类似这样的问题：数据整体是一年的按时间排序的皮大衣销售数据。如果不用随机采样策略，划分后训练集中只包括春、夏两季的用户数据，而测试集中是秋、冬季的数据。此时用春、夏季数据训练的模型明显无法很好地预测秋、冬季销售情况。

同理，如果医疗诊断系统中训练集与测试集有不同的病人年龄层分布，那也无法训练出适配所有年龄层次人的诊断模型。

3. 分层采样（Stratified Sampling）

划分数据集时的另一个常见陷阱是每种标签的数据没有均匀地被划分到训练集与测试集中。比如在医疗诊断系统中，如果将健康人群都分到了训练集，而有病况的人群都被分到了测试集，那么训练出的模型肯定会漏诊，即将有病况的人预测为健康人。

所谓的分层采样就是一种在划分训练/测试集时保持标签数据比例的采样规则，如图 1-14 所示。

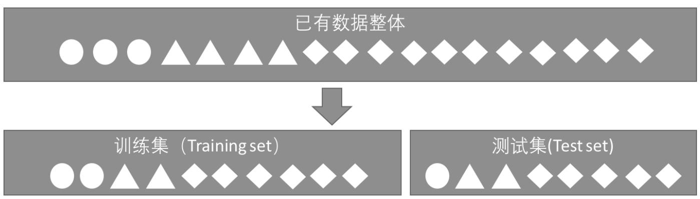


图 1-14　分层采样

图中圆形、三角形、菱形分别代表三种数据标签，分层采样的要求是将所有类型的标签等比例划分到不同数据集中。

4. 验证集（Validation set）

验证集是在某个模型的学习过程中用来调试超参数的数据集。因为大多数算法有可配

超参数（如神经网络层数、EM 类算法的最大迭代数等），所以验证集在机器学习领域也很常见，其作用如图 1-15 所示。

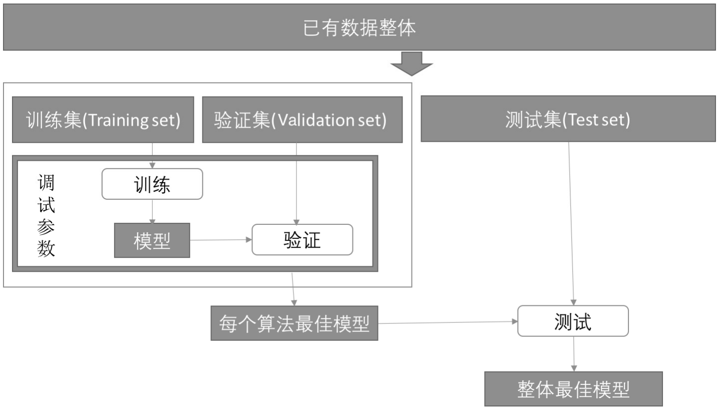


图 1-15　验证集的作用

图 1-15 中将图 1-13 的训练集又拆分成两部分，成为训练集和验证集。验证集的作用是，在调试算法参数的不断“训练-验证”迭代中验证参数的性能，以达到选择正确超参数的目的。因为验证集只在备选算法学习时内部使用，因此可以对每个备选算法选取各自独立的验证集。

如有足够多的已有数据，划分三个子集的整体数量保持如下不等式关系可以提高模型的泛化能力：

训练集数量 < 验证集数量 < 测试集数量

也就是说当从小数据集训练出的模型能够适配比其大的数据集时，才能更有把握地相信其学习到了一个普遍适用的知识。

1.4.2　交叉验证

在“训练-验证”迭代中，与使用固定的验证集相对的另外一个重要方法是交叉验证（Cross-Validation）。它是一种将数据集划分成若干子集，然后互作训练和验证的方法，其过程如图 1-16 所示。



2. 迭代训练评估

迭代	训练集	验证集	正确率（%）
1	子集1、2、3	子集4	41/50
2	子集2、3、4	子集1	30/50
3	子集3、4、1	子集2	37/50
4	子集4、1、2	子集3	43/50

3. 整体评估　整体正确率：(41+30+37+43)/200 = 75.5%

图 1-16　交叉验证过程

图中首先将训练整体划分为样本数量相等的 *n* 个子集；进行 *n* 次迭代，每次迭代取一个子集作为验证集，其他子集作为训练集，在迭代内计算验证结果；最后综合所有迭代的验证结果作为整体的模型正确率。在实践中有以下几点值得注意。

◎　子集的划分数量 *n* 可以自行定义，就是所谓的 n-fold 交叉验证。

◎　在数据量足够的情况下，n-fold 一般取值为 10。但如果模型训练时间过长则只能降低该值，比如降到 5 左右。

◎　随机采样与分层采样原则在划分子集时仍然需要遵守。

◎　同一个算法在不同 n-fold 上表现可能不一样，所以在比较不同算法的交叉验证结论时需要用相同的 n-fold 配置。

◎　交叉验证的概念同样适用于模型测试。此时直接将所有数据样本作为整体划分子集，而不是图 1-16 中的训练样本整体。

1.4.3　评估指标

上一节的图 1-16 在做交叉验证结论时使用的是整体分类正确率（Accuracy，也称准确率），但其实有时需要特别关注某个标签的分类结果。此外，连续值标签无法用准确率的方式评估。本小节讨论适用于这些目标的评估指标。

1. 整体准确率指标的缺陷

准确率的定义是：被正确预测标签的样本百分比。初看时这个指标非常公正，但在很

多场合其实无法真实体现分类模型的能力。设想一个用于在体检中诊断肺气肿的分类器，大多数的健康人是无此疾病的（假设此病的发病率是 5%）。此时如果一个分类器的准确率是 95%，那么它的疾病诊断能力如何呢？

答案很可能是：这个模型没有任何分类能力！

原因是：如果这个分类器简单地将所有病人都诊断为“健康”，由于发病总体本身只占 5%，那么一个什么都不做的分类器都可以达到 95%的准确率，如图 1-17 所示。

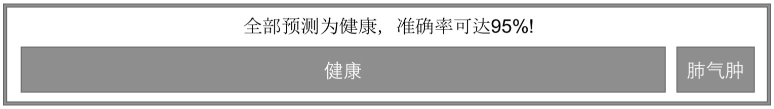


图 1-17　什么都不做的分类器可以达到很高的准确率

本例中如果单独考虑肺气肿病例的分类，那么显然该标签的准确率为零。因此，当不同标签的样本在数据集中分布不均匀时，需要独立考察模型对每个标签的分类能力。

2. 二值混淆矩阵

二值混淆矩阵（Positive/Negative Confusion Matrix）是独立考察模型对某个标签分类能力的工具，在其中引入了四个进一步考察分类器的术语，如图 1-18 所示。

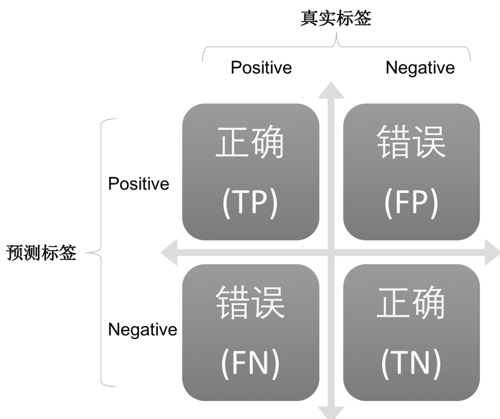


图 1-18　二值混淆矩阵引入的考察分类器的术语

它是一个 2×2 矩阵，用来表示某个标签在验证/测试集中的真实数量与预测数量的关系。图中的 Positive 指被考察标签，Negative 指除了被考察标签之外的其他标签。以肺气肿诊断举例，矩阵中的四个元素如下。

◎ TP：被考察标签被成功预测的数量，即肺气肿病人被正确查出的数量。

◎ FP：非被考察标签被预测为被考察标签的数量，即健康人被误诊为肺气肿病人的数量。

◎ FN：被考察标签未命中的数量，即肺气肿病人被误诊为健康人的数量。

◎ TN：非被考察标签被预测为非被考察标签的数量，即健康人被诊断为健康人的数量。

显然，四种情况中 TP 和 TN 是分类器成功预测的情况，FP 和 FN 是分类器预测失败的情况，其中 FP 是漏报，FN 是误报。对于只有两种标签的数据整体，准确率可用这四个元素来定义：

```
Accuracy = ( TP + TN ) / ( TP + TN + FP + FN )
```

说明：只需将矩阵的纵/横坐标的取值定义为多个可用标签，混淆矩阵就可以被用来解释所有标签的被分类能力。

3. 精确率、召回率、调和均值

已经解释过准确率不足以反映分类模型的实际效果，所以在混淆矩阵的基础上又定义了另外几个分类评估指标，常用的有如下几个。

◎ 精确率：Precision = TP / (TP + FP)

◎ 召回率：Recall = TP / (TP + FN)

◎ 调和均值：Harmonic Mean = 2TP / (2TP + FP + FN)

精确率是被预测为正例的样本中正确的比率，比如所有被诊断为肺气肿的病人中真的患有肺气肿的病人比例；召回率是正例样本中被成功检测出的比率，比如患有肺气肿的人中被正确识别出的比率；调和均值可以看成对前两者的综合。

因此，一般情况下调和均值可以直接考察模型对单个标签的分类能力。

4. 连续值指标

由于连续值标签的数量不可枚举，所以无法用之前介绍的几个指标进行考量。此时可以采用源自统计学的两个回归误差指标，如下所示。

◎　平均绝对差：$\text{MAE}(y,\hat{y})=\sum_{i=1}^{N}\left|y_i-\hat{y}_i\right|/N$，是所有被测样本的真实值与预测值的绝对差均值。

◎　平均方差：$\text{MSE}(y,\hat{y})=\sum_{i=1}^{N}(y_i-\hat{y}_i)^2/N$，即所有被测样本的真实值与预测值的平方差均值。

以上两个公式中，y 是样本真实标签值，$\hat{y}$ 是预测值，N 是验证/测试样本总数。

1.4.4　拟合不足与过度拟合

训练出的模型无法反映客观事实的两类最常见的错误是所谓的拟合不足（Underfitting）与过度拟合（Overfitting）。这两个术语均来自于统计学，分别指模型无法匹配训练数据（拟合不足）、模型匹配训练数据过度导致无法匹配测试数据（过度拟合）。图 1-19 以回归模型做了举例说明。

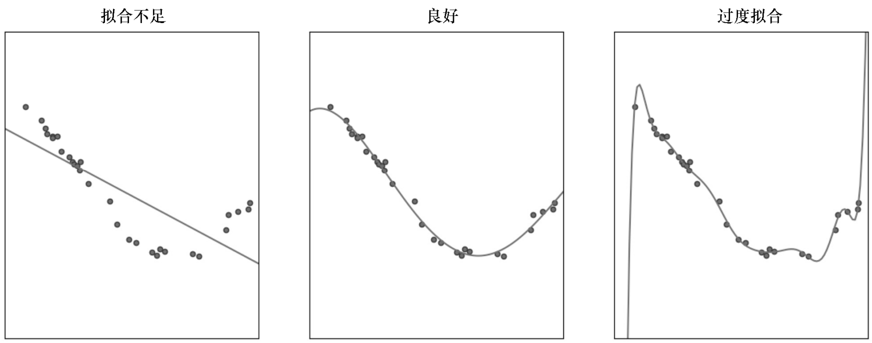


图 1-19　拟合不足与过度拟合

如图 1-19 所示是三个对相同散点集拟合的线性回归模型。该组散点取自三角函数 sin()的一个区间，并加入了一些随机干扰噪声。对这组散点理想的拟合就是还原 sin 曲线。

左侧子图是拟合不足，它将这组散点拟合成了直线，也就是其预测所有点都在该条直线上，在这种情况下即使在训练集上均方差等评估指标也会得到非常差的结果。

右侧子图是过度拟合，该图形几乎拟合了所有训练集数据点，但正因如此，它也不是一个正常的 sin 曲线拟合，对新数据尤其是位于边界附近的预测会产生比中图差得多的效果。

对于这两类错误只能通过调节算法超参数来解决，从某种程度上讲，任何机器学习算法都是围绕着如何避免这两类问题的出现而展开的。

思考：生活中有哪些拟合不足与过度拟合的例子呢？

1.5 本章内容回顾

◎ 机器学习的三个要素：明确的任务、可衡量、可训练。

◎ 虽然出发点不同，但机器学习与数据挖掘共用了大部分方法与流程。

◎ 机器学习是人工智能四大领域中的一个。另外三个领域是：自然语言处理、知识表示、自动推理。

◎ 机器学习技术的发展正影响着人工智能的另外几个领域。

◎ 目前的人工智能尚无所谓的人类自我意识。

◎ 机器学习开发的一般流程：定义问题→收集数据→多次迭代比较算法模型→选取最优模型并应用。

◎ 机器学习算法的三大类：有监督学习、无监督学习、强化学习。另有一些综合工具：如隐马尔可夫模型、贝叶斯网络、文档主题模型。

◎ 有监督学习可分为两大类：分类与回归。

◎ 无监督学习主要包括聚类、降维等。

◎ 强化学习的目标是训练一个在环境中可成长的智能代理，是最接近人工智能原本目的的一个学习方法。

◎ 学习中常用的三个数据集：训练集、验证集、测试集。

◎ 交叉验证概念与方法。

◎ 几个评估指标：准确度、精确度、召回率、调和均值、平均绝对差、平均方差。前四个用于离散值标签问题，后两个用于连续值标签问题。

◎ 拟合不足是模型无法匹配训练数据的问题。过度拟合是模型过于匹配训练数据、同时导致无法匹配测试数据的问题。

2 第 2 章 Python 基础工具

Python 在 IEEE 于 2017 年 7 月发布的计算机语言榜单中排名第一，在 2018 年 7 月的 TIOBE 排行榜中名列第三。数据科学与云计算是 Python 应用的重要领域，本书的所有机器学习实践部分都使用 Python 语言开发。本章介绍三种 Python 数据处理工具，为后续实践打下技术基础，只希望了解学习场景与算法原理的读者可以跳过本章直接从第 3 章开始阅读。本章主要内容如下。

◎ Numpy 应用：数据封装工具，学习其中的标量、向量、索引、遍历、常用操作。

◎ Matplot 应用：用于展示数据的工具，学习散点图、组合图、图像等的基本绘制方法。

◎ Scipy 应用：用于科学计算的工具，比如微积分、插值、傅里叶变换、矩阵等。

2.1 Numpy

几乎所有的 Python 科学计算库都使用 Numpy 封装数组与矩阵运算，其重要性对于机器学习开发者来说不言而喻。本节不是 Numpy 的完全参考手册，但是引入足够的必要知识能够帮助读者顺利阅读本书代码，并使读者有能力在需要时快速查找在线手册。这个目标同样适用于后面的其他工具。

首先，通过 pip3 命令安装 Numpy：

```
#pip3 install numpy
#python3
>>> import numpy as np
>>> np.__version
1.14.5
```

在使用 Numpy 的程序中笔者习惯性地将其重命名为 np，本书后续代码沿用这一约定。目前为止 Numpy 的 pip 稳定版是 1.14.5，也是本书学习与使用的版本。

2.1.1 Numpy 与 Scipy 的分工

Numpy 与 Scipy 都是 Python 的基础运算库，在学习之前有必要明确两者的异同点。按照 Scipy 官方网站的定义，在理想情况下，Numpy 应该只包含多维数组数据结构本身和一些围绕其进行的基本操作：读取、排序、变形等；而 Scipy 是利用 Numpy 的基础数据结构进行数学运算，比如线性代数、概率分布、傅里叶运算等。

但由于 Numpy 是由 Numeric、numarray 等开源库逐步发展而来的，出于历史兼容的原因 Numpy 也开发了相当多的数学运算封装，在这些运算中很多是与 Scipy 互相重复的，比如 Numpy 和 Scipy 都有线性代数运算函数。

思考：开发者如何选择 Numpy 和 Scipy 呢？

对于这个问题可以考虑如下几个方面：

◎ 用 Numpy 管理数据多维数组结构。

◎ 虽然 Scipy 的多数运算库在 Numpy 中也有实现，但通常 Scipy 中的版本提供了更强大的功能，所以对新项目开发来说，选择 Scipy 进行除加减乘除外的其他高级数学运算。

◎　很多历史开源代码选择了Numpy的数学运算，其调用形式与Scipy中的方式类似。所以掌握 Scipy 的开发者可以顺利读懂 Numpy 的数学运算，无须特别学习 Numpy 中的庞大运算库。

◎　对于新运算功能的开发往往被包含在 Scipy 中，而不是 Numpy。

出于这几点，本节围绕 Numpy 数组类型 ndarray 讲解多维数组的数据结构和基本操作，而将数学运算方面的大量内容放在 Scipy 中讲解。

2.1.2　ndarray 构造

Numpy 是一组围绕 ndarray 数组及其相关运算定义的函数库。从功能上说，ndarray 数组提供与 Python 原生列表类型（list）几乎相同的功能，但使用 ndarray 可以提高程序的开发和运行效率。比如，用 Python list 做两个数组元素间乘法的代码如下：

```
a = [1, 2, 3, 4, 5, 6]
b = [1, 4, 9, 16, 25, 36]
c = []
for i in range( len(a)):
    c.append(a[i] * b[i])
```

以上代码中需要用一个循环执行逐个元素的相乘。而使用 Numpy 数组时的代码是：

```
a = np.array([1, 2, 3, 4, 5, 6])
b = np.array([1, 4, 9, 16, 25, 36])
c = a * b
```

以上代码直接使用 ndarray 的乘法运算进行元素相乘，不但简洁易懂，而且由于 Numpy 本身主要由 C 语言编写，因此在代码执行效率上也快很多。这个速度优势在数组元素不多时体现不明显，但对于图像处理等程序动辄几十万的数组长度来说优势是巨大的。

1.　给定值生成 ndarray

ndarray 是 Numpy 的核心对象，numpy.array()函数是它最简单的构造方式。ndarray 的完整名称是 numpy.ndarray，容易混淆的是，它的对象描述名称是 array，比如：

```
>>> a = np.array([1, 2, 3, 4, 5, 6])          # 初始化函数
>>> type(a)
<class 'numpy.ndarray'>                       # 类型完整名称
```

```
>>> a
array([1, 2, 3, 4, 5, 6])                          # 对象描述
```

说明：在不同的书籍与文章中，有时 ndarray 类型也被称为 Numpy 数组。

与 Python 数组不同的是，在 ndarray 中一个数组的所有元素必须是相同数据类型，因此在初始化列表中发现不同类型的元素时需要做类型提升：

```
# 只要元素中有一个是浮点数，其他元素也被提升为浮点数
>>> np.array([1, 2, 3, 4, 5, 6.1])
array([ 1. ,  2. ,  3. ,  4. ,  5. ,  6.1])      #1. 是 1.0 的简写方式
```

其他一些技巧还包括：

```
>>> np.array([[1, 2, 3], [4, 5, 6]])              # 多维数组
array([[1, 2, 3],
[4, 5, 6]])

>>> np.array([1, 2, 3], dtype=complex)            # 定义类型
array([ 1.+0.j,  2.+0.j,  3.+0.j])
```

2. 快速生成构造

可以用 empty()、eye()、ones()、zeros()等函数构造未初始化、由 0 和 1 组成的数组或矩阵：

```
>>> np.empty(3)                                    # 用随机值构造数组，速度最快
array([  4.94065646e-324,   9.88131292e-324,   1.48219694e-323])

>>> np.empty([2, 3])                               # 构造 2×3 矩阵，
array([[  1.09385605e-303,   1.09385187e-303,   1.09385326e-303],
[  1.09402988e-303,   9.12411861e-063,   7.78003736e-299]])

>>> np.eye(3, 4)                                   # 构造只有对角线元素为 1 的矩阵
array([[ 1.,  0.,  0. ,  0.],                      # 注意：可以构造非方阵
       [ 0.,  1.,  0. ,  0.],
[ 0.,  0.,  1. ,  0.]])

>>> np.zeros(4)                                    # 构造全 0 数组
array([ 0.,  0.,  0. ,  0.])

>>> np.ones([2, 3, 4])                        # 与 zeros()对应，构造全 1 多维数组
```

```
array([[[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]],

       [[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]]])

>>> >>> np.tri(3)                              # 下三角单元矩阵
array([[ 1.,  0.,  0.],
       [ 1.,  1.,  0.],
       [ 1.,  1.,  1.]])
```

以上代码中用整数列表指定初始化的数组维数，比如 np.ones（[2, 3, 4]）用于初始化 2×3×4 的三维数组。这个整数列表就是 Numpy 中所谓的 shape 属性，很多 Numpy 函数都用 shape 作为参数。当给 shape 参数传入整数时（比如 *n*），其作用与传入列表[*n*,]一样，都是指一个一维数组。

3. 插值构造

arange()、linspace()、logspace()等函数可以在给定的取值范围内用步长、比值等规则构造数组序列，比如：

```
>>> np.linspace(1, 20, num=7)                  # 在 1～20 之间生成 7 个元素
array([  1. , 4.16666667, 7.33333333,  10.5 ,
       13.66666667,  16.83333333,  20. ])
>>> np.arange(1, 20, step=7)                   # 在 1～20 之间每 7 个值生成一个元素
array([ 1,  8, 15])
>>> np.logspace(1, 20, num=7, base=10)         # 在 10^1～10^20 之间取 7 个指数值
array([  1.00000000e+01,   1.46779927e+04,   2.15443469e+07,
         3.16227766e+10,   4.64158883e+13,   6.81292069e+16,
         1.00000000e+20])
```

4. 随机采样构造

除了普通的随机采样，Numpy 还提供了大量的按照某种分布随机产生 ndarray 元素的方法，比如：

```
>>> np.random.rand(2, 5)                   # 随机生成一个 2×5 的二维数组
```

```
array([[ 0.34836619,  0.36164538,  0.67479146,  0.96182738,  0.93734632],
[ 0.56939177,  0.36255749,  0.07442314,  0.03607615,  0.89874155]])

# 按均值 1、标准差 0.5 的正态分布生成 5 个随机数
>>> np.random.normal(1, 0.5, size=5)
array([ 0.78435269,  2.35796338,  0.78268042,  1.43587983,  0.47725419])
```

类似 random.normal()的按概率分布生成数组的函数还有很多，比如 random.beta()、random. dirichlet()、random.poisson()等。

Numpy 中还有很多数组构造函数，比如从其他源对象转换成 ndarray 的 asarray()、asmatrix()、copy()、frombuffer()、fromfile()、fromiter()等，直接构建矩阵的 diag()、tril()、triu()、vander()等，这里不再一一举例。

提示：在 Numpy 中还定义了一个 ndarray 子类 matrix，即矩阵。它继承了 ndarray 的所有特性，只是维度被固定为二维，本书不再对其进行单独讨论。

2.1.3 数据类型

ndarray 中的元素类型没有使用 Python 的类型系统，而是提供了一套更丰富的类型系统。开发者既可以使用 Numpy 已有的基本数据类型（dtype），也可以自定义新的元素数据类型。

1. 基本数据类型

Numpy 的所有类型可以通过如下代码获得完整列表：

```
>>> type_dict = np.typeDict.values()
>>> sorted([ x.__name__ for x in list(set(type_dict))])
['bool_', 'bytes_', 'complex128', 'complex256', 'complex64', 'datetime64',
'float128', 'float16', 'float32', 'float64', 'int16', 'int32', 'int64', 'int64',
'int8',  'object_',  'str_',  'timedelta64',  'uint16',  'uint32',  'uint64',
'uint64','uint8', 'void']
```

对不同大小的数值定义不同的类型，使得开发者有机会选择恰当的元素大小，以免造成不必要的空间浪费。在 ndarray 的构造函数中可以用 dtype 参数指定元素类型，比如：

```
>>> np.array([1, 2.334, 3.1415926], dtype= 'uint8')
array([1, 2, 3], dtype=uint8)
```

对于已有的 ndarray 对象，也可以通过读取其 dtype 属性获得元素类型。

```
>>> np.array([1, 2.334, 3.1415926]).dtype
dtype('float64')
```

使用 astype()函数可以转换 ndarray 数组的元素类型：

```
>>> a = np.array([1, 2.334, 3.1415926], dtype= 'uint8')
>>> a.astype('complex')                          # 转换为复数类型
array([ 1.+0.j,  2.+0.j,  3.+0.j])
```

2. 自定义数据类型

开发者可以利用基本数据类型定义新的组合数据类型。比如，如果元素有两个数据段 age 和 gender：

```
>>> type_person = np.dtype([('age', 'uint8'), ('gendar', 'bool')])
>>> a = np.array([(8, True), (16, False), (10, True)], dtype = type_person)
>>> a
array([( 8,  True), (16, False), (10,  True)],
dtype=[('age', 'u1'), ('gendar', '?')])
```

代码中建立了新的组合数据类型 type_array，该类型的两个子字段分别为 uint8 和 bool 类型。然后初始化了该类型的一个一维数组 a，可以通过下标索引访问子元素：

```
>>> a[0]
(8,  True)
>>> a[0]['age']
8
```

2.1.4　访问与修改

Numpy 提供了切片、下标索引、循环迭代等多种方式用于访问和修改（slicing、indexing、iteration、unique）数组元素。

1. 切片

Numpy 在 ndarray 中保留了 Python 对列表变量的切片操作（slicing）用来读取数组内容。切片的基本语法是一个 i:j:k 索引器，其中 i 是起始索引，j 是结束索引，k 是步长。比如：

```
>>> x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> x[2:7:2]                              # 从索引 2 开始、每两步读取一个，到 7 为止
```

```
array([2, 4, 6])
```

i、j、k 的取值允许是负值，其含义是从倒数索引，比如：

```
>>> x[-7:-1:]                                    # 从倒数第 7 个开始，到倒数第 1 个
array([3, 4, 5, 6, 7, 8])

>>> x[7:1:-2]                                    # 步长为负，从后向前读
array([7, 5, 3])
```

切片操作不仅可以读取数组，还可以用来直接修改元素内容：

```
>>> x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> x[5:7] = [100, 101]
>>> x
array([  0,   1,   2,   3,   4, 100, 101,   7,   8,   9])
```

2. 下标访问

除切片外，Numpy 还允许直接使用下标列表读取 ndarray 数组的内容：

```
>>> x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> x[[3, -1, -2]]                               # 读取索引为 3, -1, -2 的元素
array([3, 9, 8])
```

也可以用布尔类型下标列表访问：

```
# 读取 True 所在位置元素
>>> x[[True, False, False, True, True, False, False, False, False, True]]
array([0, 3, 4, 9])
```

其中布尔列表的长度必须与数组 x 长度相同。

3. 迭代

对于一维数组，可以直接迭代 ndarray 以访问所有元素：

```
>>> for i in x:
...     print(i, end=', ')

0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

对于多维数组，可以使用 ndenumerate()函数在迭代的同时获得元素下标：

```
>>> x = np.array([[1, 2, ], [5, 6, ], [9, 10,]])        # 3×2 的二维数组
```

```
>>> for index, value in np.ndenumerate(x):
...     print(index, ": ", value)
...
(0, 0) :  1
(0, 1) :  2
(1, 0) :  5
(1, 1) :  6
(2, 0) :  9
(2, 1) :  10
```

4. 唯一列表

通过 unique()函数可以生成非重复元素数组，还可以返回元素正向与反向构造的索引、元素计数等，比如：

```
a = np.array([1, 2, 6, 4, 2, 3, 2])
>>> np.unique(a, return_index=True, return_inverse=True,
        return_counts=True)
(array([1, 2, 3, 4, 6]), array([0, 1, 5, 3, 2]), array([0, 1, 4, 3, 1, 2,
1]), array([1, 3, 1, 1, 1]))
```

其中第一个返回值是结果数组，第二个返回值是结果数组中每个元素在原数组中的索引，第三个返回值是原数组中每个元素在结果数组中的索引，第四个返回值是结果数组中每个元素在原数组中的计数。

2.1.5　轴

不仅是 Numpy，轴（axis）的概念在 Matplot 等其他 Python 科学计算包中也随处可见。在 Numpy 中它是很多排序、变形操作的基础。本小节用举例的方式来讨论轴的用法。

1. 排序

Numpy 排序函数 sort()原型如下：

```
numpy.sort(a, axis=-1, kind='quicksort', order=None)
Return a sorted copy of an array.
Parameters:
a : array_like
Array to be sorted.
axis : int or None, optional
```

```
    Axis along which to sort. If None, the array is flattened before sorting.
The default is -1, which sorts along the last axis.
    kind : {'quicksort', 'mergesort', 'heapsort'}, optional
    Sorting algorithm. Default is 'quicksort'.
    order : str or list of str, optional
    When a is an array with fields defined, this argument specifies which fields
to compare first, second, etc. A single field can be specified as a string, and
not all fields need be specified, but unspecified fields will still be used, in
the order in which they come up in the dtype, to break ties.
    Returns:
    sorted_array : ndarray
    Array of the same type and shape as a.
```

其中 a 是输入数组；axis 是轴，可以取值 None 或者整型；另外两个参数 kind 和 order 指定排序方法和排序字段。理解 axis 的含义是灵活运用该函数的关键。

轴概念的产生是由于需要处理多维数组，与维度（ndim）息息相关。Numpy 维度概念与 Python 列表中的维度相同，比如：

```
>>> a = np.array([1, 2, 3, 4, 5])                    #一维数组
# 二维数组，即矩阵
>>> b = np.array([[1, 2, 1], [3, 4, 3], [5, 6, 5], [4, 5, 7]])
# 三维数组
>>> c = np.array([[[1, 2], [11, 12], [5, 6]], [[-1, -2], [13, 14], [15, 3]]])
>>> c.ndim                                           #读取已有数组维度
3
>>> c.shape                                          #读取形状
(2, 3, 2)
```

轴的概念就是指从零开始的“数组的第几个维度”，sort()函数中的 axis 参数用于指定按哪个维度排序，比如图 2-1 是二维数组的排序示例。

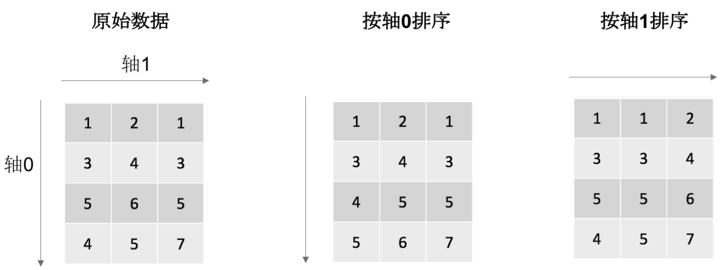


图 2-1　轴与 sort()函数

当对 sort()的 axis 设置 None 时，函数先将多维数组平铺，然后返回所有元素总的结果：

```
>>> c = np.array([[[1, 2], [11, 12], [5, 6]], [[-1, -2], [13, 14], [15, 3]]])
>>> np.sort(c, None)
array([-2, -1,  1,  2,  3,  5,  6, 11, 12, 13, 14, 15])
```

2. 求和

另一个以 axis 作为参数的例子是求和函数 sum()：

```
>>> c = np.array([[[1, 2], [11, 12], [5, 6]], [[-1, -2], [13, 14], [15, 3]]])
>>> np.sum(c, None)
79
>>> c
array([[[ 1,  2],
        [11, 12],
        [ 5,  6]],

       [[-1, -2],
        [13, 14],
        [15,  3]]])
>>> np.sum(c, 1)
array([[17, 20],                # 17 = 1 + 11 +5,    20 = 2 + 12 +6
       [27, 15]])               # 27 = -1 + 13 + 15,   15 = -2 + 14 +3
```

一个 2×3×2 的三维数组在对轴 1 进行聚合操作后变成 2×2 的二维数组。

3. 转置

数组转置操作的函数原型：

```
numpy.transpose(a, axes=None)
```

在 Numpy 中数组转置操作是维度顺序的调整。比较好理解的是二维数组的转置：

```
>>> b = np.array([[1, 2, 1], [3, 4, 3], [5, 6, 5], [4, 5, 7]])
>>> np.transpose(b)
array([[1, 3, 5, 4],
       [2, 4, 6, 5],
[1, 3, 5, 7]])
```

对于多维数组的转置则可能有多种变换方法，比如轴的顺序原来是 0/1/2，转置后可以变换为 1/2/0、2/0/1 或 1/0/2 等。这时就可以通过 axes 参数来指定目标轴的顺序，具体实现代码如下。

```
>>> c                                              # 原始数组 2×3×2
array([[[ 1,  2],
        [11, 12],
        [ 5,  6]],

       [[-1, -2],
        [13, 14],
        [15,  3]]])
# 转置轴 0 与 2，维度仍是 2×2×3，但内容有变化
>>> np.transpose(c, axes = (2, 1, 0))
array([[[ 1, -1],
        [11, 13],
        [ 5, 15]],

       [[ 2, -2],
        [12, 14],
        [ 6,  3]]])
>>> np.transpose(c, axes = (1,0, 2))               # 转置 0 与 1，维度变为 3×2×2
array([[[ 1,  2],
        [-1, -2]],

       [[11, 12],
        [13, 14]],

       [[ 5,  6],
        [15,  3]]])
```

注意：axes 是 axis 的英文复数形式，在查找手册时可以根据名字就判断出为该参数传递数值列表还是单个数值。

2.1.6 维度操作

本小节学习常用的 ndarray 形状与维度操作（shape and dimension）。

1. 变形（changing shape）

最典型的变形操作是 reshape()，它可以将已有数组变换成任何维度的数组：

```
>>> b = np.array([[1, 2, 1], [3, 4, 3], [5, 6, 5], [4, 5, 7]])
>>> b.shape
```

```
(4, 3)
>>> np.reshape(b, [2, 2, 3])                # 将 4×3 数组变换为 2×2×3
array([[[1, 2, 1],
        [3, 4, 3]],

       [[5, 6, 5],
        [4, 5, 7]]])
```

可以将变形理解为先把所有元素平铺再按需求组装的过程，需要注意变换后的数组元素总数必须与原数组中的相同，本例中即为 4×3==2×2×3。另外还有函数 ravel()、ndarray.flat()、ndarray.flatten()等对数组进行平铺的变形函数。

还可以通过 flip()函数反转某个轴的数组内容：

```
e = np.array([[1, 2, 1, 2], [3, 4, 3, 4], [ 5, 6, 5, 6]])
>>> np.flip(e, axis =0 )                        # 反转轴 0
array([[5, 6, 5, 6],
       [3, 4, 3, 4],
[1, 2, 1, 2]])
```

与 flip()类似的还有适用于矩阵数组的 fliplr()与 flipud()函数。它们不是反转某个轴，而是按矩阵对角线进行反转。

2. 维度操作（changing number of dimensions）

此类操作允许对数组增加或减少维度，其中 atleast_1d()、atleast_2d()、atleast_3d()用于将任意对象置于有至少 1 个维度的数组中，通常用于数组初始化，比如：

```
>>> np.atleast_3d(3.14)                         # 将数值放入三维数组中
array([[[ 3.14]]])
```

expand_dims()用于在指定轴增加维度：

```
>>> d = np.array([[1, 2, 1], [3, 4, 3]])        # 原始数组形状是 2×3
>>> np.expand_dims(d, 1)                        # 在轴 1 处插入一个维度
array([[[1, 2, 1]],                             # 新数组形状变为 2×1×3
       [[3, 4, 3]]])
```

squeeze()函数可以看成 expand_dims()的反函数，它移除长度为 1 的维度：

```
>>> e = np.array([[[1, 2, 1], [3, 4, 3]]])      # 原始数组形状 1×2×3
>>> np.squeeze(e, axis = 0)                     # 去除轴 0
array([[1, 2, 1],                               # 新数组形状 2×3
```

```
       [3, 4, 3]])
```

2.1.7 合并与拆分

Numpy 中有一组进行合并与拆分（joining and splitting）ndarray 数组的函数，它们是拼接函数 stack()、concatenate()、column_stack()等，拆分函数 split()、array_split()等。

1. 合并

column_stack()是将多个一维数组合并成二维数组/矩阵的函数：

```
>>> a = np.array((1,2,3, 4))
>>> b = np.array((2,3,4,5))
>>> c = np.array((3,4,5,6))
>>> np.column_stack((a,b, c))       # 三个长度为 4 的一维数组合并为 4×3 的矩阵。
array([[1, 2, 3],
       [2, 3, 4],
       [3, 4, 5],
       [4, 5, 6]])
```

concatenate()与 stack()进行更灵活的拼接：

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 6], [7, 8]])
>>> np.concatenate((a, b), axis = 0)       # 接续，维度不变
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
>>> np.stack((a, b), axis = 0)             # 建立新的维度
array([[[1, 2],
        [3, 4]],
       [[5, 6],
        [7, 8]]])
```

以上代码给出了 concatenate()与 stack()的区别：concatenate()是在已有维度上拼接，本例中输入数组与输出数组都是二维；stack()是在拼接的同时建立新的维度，其中 stack()要求两个输入数组的形状必须相同。另外与 stack()类似的函数还有 vstack()、hstack()、dstack()等，分别在指定维度拼接数组。

2. 拆分

数组拆分的典型函数是 splite()，其原型为：

```
split(a, indices_or_sections[, axis])
```

其中 a 是输入数组，indices_or_sections 是划分方式，a 既可以设为一个整数也可以是一个数列，理解 indices_or_sections 是运用拆分的关键，举例如下：

```
>>> x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8])
>>> np.split(x, indices_or_sections=3)            # 将 x 均匀地分为 3 份
[array([0, 1, 2]), array([3, 4, 5]), array([6, 7, 8])]

>>> y = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8])
>>> np.split(x, indices_or_sections=[2, 3, 6]) # 分别在 2、3、6 处截断源数组
[array([0, 1]), array([2]), array([3, 4, 5]), array([6, 7, 8])]
```

axis 参数用于指定拆分的轴，比如：

```
# 二维数组，3×4 数组
>>> e = np.array([[1, 2, 1, 2], [3, 4, 3, 4], [ 5, 6, 5, 6]])
>>> np.split(e, 2, 1)                    # 按照轴 1 进行拆分，变为两个 3×2 数组
[array([[1, 2],
       [3, 4],
[5, 6]]),
 array([[1, 2],
       [3, 4],
       [5, 6]])]
```

另外 Numpy 中还有 hsplit()、vsplit()、dsplit()等函数用于指定轴的数组拆分，比如 vstack()与 split(axi=1)的作用相同。

2.1.8　增与删

Numpy 通过 delete()、insert()、append()三个函数进行元素增删（adding and remove）操作：

```
>>> e = np.array([[1, 2, 1, 2], [3, 4, 3, 4], [ 5, 6, 5, 6]])
>>> np.delete(e, 1, axis = 0)              # 删除轴 0 位置为 1 的一行元素
array([[1, 2, 1, 2],
[5, 6, 5, 6]])
```

```
>>> np.insert(e, 2 , [0, 0, 1], axis=1)    # 在轴 1 的位置 2 插入一列元素
array([[1, 2, 0, 1, 2],
      [3, 4, 0, 3, 4],
      [5, 6, 1, 5, 6]])
```

append()与 insert()类似，但是只能在轴的末尾插入，无法指定位置。

2.1.9 全函数

Numpy 中的 ufunc 是一组能对数组中每个元素进行批量操作的全函数。ufunc 内部通过 C 语言来实现，所以速度非常快。举例如下：

```
>>> a = np.array([1, 3, 5, 7, 9])
>>> b = np.array([0, 2, 4, 6, 8])
>>> np.add(a, b)
array([ 1,  5,  9, 13, 17])
```

其中 np.add()就是一个 ufunc，它实现了两个数组中每个元素的加法。开发者还可以直接对 ndarray 调用算数运算符使用 ufunc，比如 np.add(a, b)与 a×b 等价，类似的情况还有 np.subtract()、np.multiple()、np.divide()等。

ufunc 的成员函数非常多，它们可以分为如下几个大类（在用到的时候可以查找在线手册）。

◎ 算数运算：比如 pow()、mod()、fabs()、log()、exp()、sqrt()等。

◎ 三角函数：sin()、cos()、tan()、arcsin()等。

◎ 比特操作：bitwise_and()、bitwise_or()、invert()、bitwise_xor()、left_shift()等。

◎ 比较：greater()、greater_equal()、less()、not_equal()、logical_and()等。

◎ 浮点运算：isfinit()、fmod()、floor()、ceil()等。

这些函数与同名的 Python 库函数意义相同，只是 ufunc 函数可以批量操作数组元素。

2.1.10 广播

ufunc 函数的使用很自然地引出了另外一个话题：当两个操作出现被操作数组在维度与形状上不一致的情况时该如何处理。Numpy 定义了处理这种情况的一套机制，即广播（broadcasting）。广播机制可以通过四条规则来描述。

◎ 当多个输入数组的维度不相同时，小维度的数组会补齐到大维度，被增加的那个轴长度被设为 1。比如两个输入数组分别是一维和二维时，一维输入数组也会被转换为二维数组。

◎ 输出数组中各个维度的长度是输入数组中各维度的最大值。

◎ 输入数组的每个轴上要么长度与输出数组相匹配，要么长度为 1，否则计算会出错。

◎ 输入数组中长度为 1 的轴中的数据将被用作与其他输入数组该轴上的所有数据进行计算。

上述规则使得计算不同形状的 ndarray 非常方便，笔者认为广播能力是 Numpy 能够如此被广泛应用的最主要原因，灵活运用广播规则正是提高 Python 大数据计算开发效率的关键所在。通过如下例子解释上述规则：

```
>>> a = np.array([[1, 3, 5], [7, 9, 11]])
>>> a*3                          # 数组与整数相乘 （规则 1、4）
array([[ 3,  9, 15],
[21, 27, 33]])

>>> b = np.array([[1,], [5,]])
>>> a*b                          # 2×3 数组与 2×1 数组相乘，得到 2×3 数组（规则 2、3）
array([[ 1,  3,  5],
       [21, 27, 33]])
>>> c = np.array([[1, 2], [3, 4],])
>>> a*c                          # 2×3 数组与 2×2 数组相乘，出错（规则 3）
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (2,3) (2,2)
```

2.2 Matplot

Matplot 是 Python 的一个绘图库，由于其面向对象的设计，并基于跨平台的 Tkinter、GTK+等 GUI 工具包而得到广泛应用。Matplot 在机器学习中被用来演示算法原理或数据分布，也是学习下一节 Scipy 数学工具包的基础。如下命令可以安装最新的 Matplot 发布版：

```
#pip3 install matplotlib
#python3
>>> import matplotlib
```

```
>>> matplotlib.__version
2.2.2
```

截至目前，matplotlib 的 pip 发布版本是 2.2.2。本节先介绍 Matplot 基本绘图流程，然后学习几个常用绘图模块。

2.2.1 点线图

模块 matplotlib.pyplot 是 Matplot 中的绘图接口，它提供了与 MATLAB 相近的绘图调用方式。一个典型的 Matplot 绘图由 Axes、Axis、Legend、Line 等要素构成。Matplot 图的基本构成如图 2-2 所示。

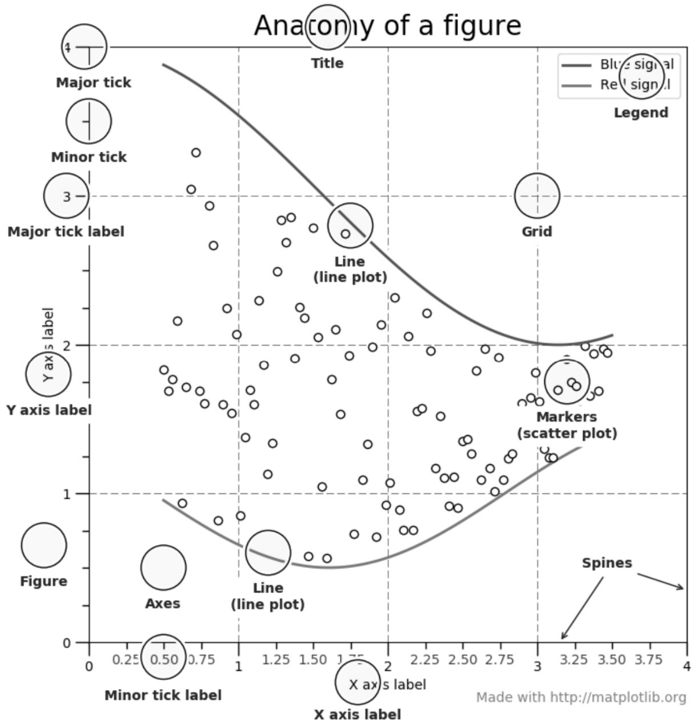


图 2-2　Matplot 图的基本构成（摘自 Matplot 官网）

这是在一个坐标系中的点线图，如下要素是开发者使用 Matplot 经常打交道的概念。

◎　Axes：坐标系，是所有 Matplot 图的基本元素，包含了图 2-2 中的整个区域，开发者可以想象它是一块有坐标的场地，可以放置其他元素。

◎　Grid：坐标系上的网格背景。

◎　Title：每个坐标系可以有一个 Title，默认放置在坐标系顶端。

◎　Legend：图例，每个坐标系可以有一个，用来提示图中各数据的文字描述。

◎　Axis：坐标轴，二维坐标系有两个坐标轴，横向的 x 轴和纵向的 y 轴。

◎　Tick：坐标轴上的刻度。出于美观和易于阅读的目的，Matplot 提供了大小两种粒度的刻度：major tick 和 minor tick。刻度上的文字被称为 tick label。

◎　Spine：边框，默认在坐标系四周都显示边框。

◎　Scatter、Line：点线数据是图的内容所在，它们是对数组数据（ndarray）的直接绘制。

下面通过一个抛物线的实例代码来演示如何绘制 Matplot 图形。

1. 数据

如下代码生成抛物线 $y=x^2+1$ 上的 20 个点：

```
import numpy as np
>>> x = np.linspace(-5, 5, 20)              # x 轴的 20 个坐标
>>> y = x**2 + 1                            # 按抛物线公式计算 y 轴坐标
```

2. 绘制

只需三行代码就可以绘制出如图 2-3 所示的抛物线模型：

```
>>> import matplotlib.pyplot as plt         # import 绘图接口
>>> plt.plot(x, y)                          # 绘制数据
>>> plt.show()                              # 显示绘图
```

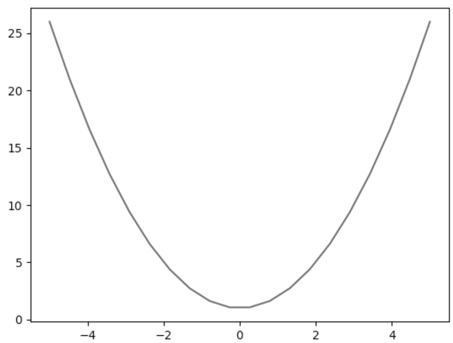


图 2-3　抛物线模型

Matplot 自动绘制了一些元素：坐标系、根据数据取值范围产生的刻度等。plot()函数的作用是绘制数据点和相邻点之间的连线，如果只想绘制数据点，可以使用 scatter()函数。Matplot 的任何绘制函数都需要在调用 show()函数后才能显示，后续不再强调。

3. 定制点与线

plot()函数的参数除了数据本身，还可以定制点线的属性，包括线段类型、颜色，点的形状、大小等，比如：

```
>>> plt.plot(x, y,                          # 输入数据
        color='green',                      # 点之间线段的颜色：绿色
        linestyle='dashed',                 # 线段的类型：虚线
        marker='o',                         # 数据点的绘制方式：圆圈
        markerfacecolor='blue',             # 数据点的颜色：蓝色
        markersize=12)                      # 数据点大小：12 points
```

以上定制点与线的结果如图 2-4 所示。

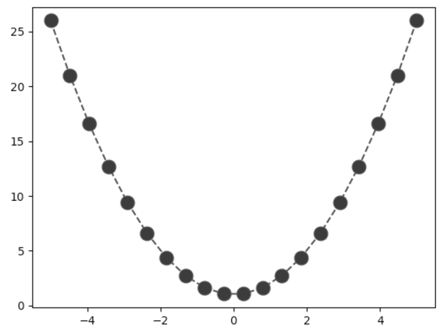


图 2-4 定制点与线

代码中的两个颜色属性可以取值常用的英文颜色，linestyle 和 marker 的常用取值如表 2-1 所示。

表 2-1 plot()参数常用取值

linestyle 取值	解释	marker 取值	解释
solid	实线	.	点
dashed	虚线	o、*	圆圈、星号
dashdot	点线相间	+、x	加号、差号
dotted	点线	v、>、<、^	下、右、左、上三角

续表

linestyle 取值	解释	marker 取值	解释
None	无线段	D、d	大、小菱形
		|、_	竖线、横线

4. 简写定制

如果只需定制点类型、线类型、颜色，plot()函数提供了一种简写的调用方式，比如：

```
>>> plt.plot(x, y, "r:x")                        # 红色、点状线段、X 号数据点
```

其绘制结果如图 2-5（左）所示。这种简写的调用方式允许一次传入多组数据，比如：

```
>>> plt.plot(x, y, "r:x",                        # 第一组数据
          x+3, y, "b-D",                         # 第二组数据
          x+6, y, "y--_")                        # 第三组数据
```

其绘制效果如图 2-5（右）所示。

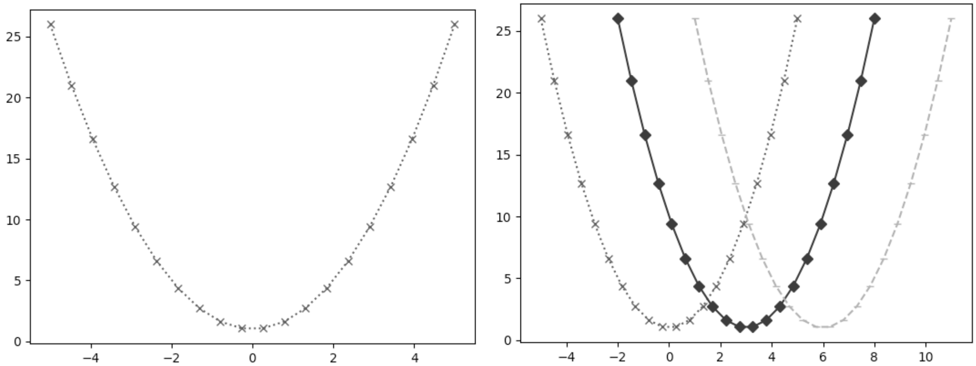


图 2-5　定制点与线

在代码的简写定制参数中用了几种线段和颜色符号，其含义如表 2-2 所示。

表 2-2　线段与颜色简写含义

线段简写	解释	颜色简写	解释
-	实线	b	blue，蓝色
--	虚线	g	green，绿色
-.	点线相间	r	red，红色
:	点线	c	cyan，青色

续表

线段简写	解释	颜色简写	解释
		m	magenta，洋红色
		y	yellow，黄色
		k	black，黑色
		w	white，白色

5. 坐标系定制

到目前为止坐标系上的刻度由 Matplot 自动生成，开发者可以用 xticks()和 yticks()函数修改两个坐标轴上的显示刻度，比如：

```
>>> plt.xticks([-4, -3, -1])                          #设置 x 轴上的刻度值
>>> plt.yticks([2, 5, 10, 20],                        #设置 y 轴上的刻度值
        ["two", "five", "ten", "twenty"])             #设置 y 轴上刻度的显示文本
```

上述代码在显示后的效果如图 2-6 所示。

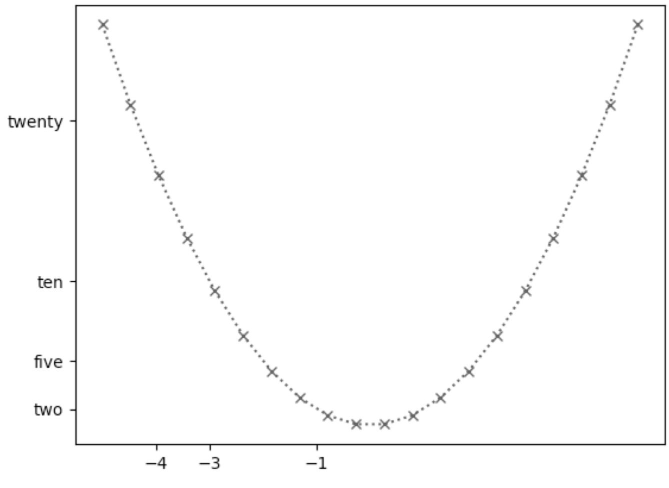


图 2-6　刻度定制效果

6. 图例与提示

当一次绘图包含多组数据时，通常需要用文本提示读者各组数据的名称，也就是图例（legend）；而提示（annotation）是用于描述图中个别关键点的文本，代码如下。

```
    >>> plt.plot(x, y, "r--x", label = "base")                # 用 label 参数
设置数据文本描述
    >>> plt.plot(x+2, y, "g-o", label = "moved")
    >>> plt.legend(loc='lower right')                     # 配置图例的显示位置
```

上述代码将在如图 2-7（左）所示的右下角显示图例。可用的图例位置还包括“upper right”“upper left”“lower left”“lower right”“center left”等，其含义显而易见。对于个别关键数据点的 annotation 示例：

```
    >>> plt.annotate('sample point',                      # 提示文本
            xy=(x[2], y[2]),                              # 提示数据点坐标
            xytext=(0, 22),                               # 提示文本显示的位置
            arrowprops=dict(facecolor='black', shrink=0.05))    # 箭头绘制属性
```

以上 annotation 效果如图 2-7（右）所示。

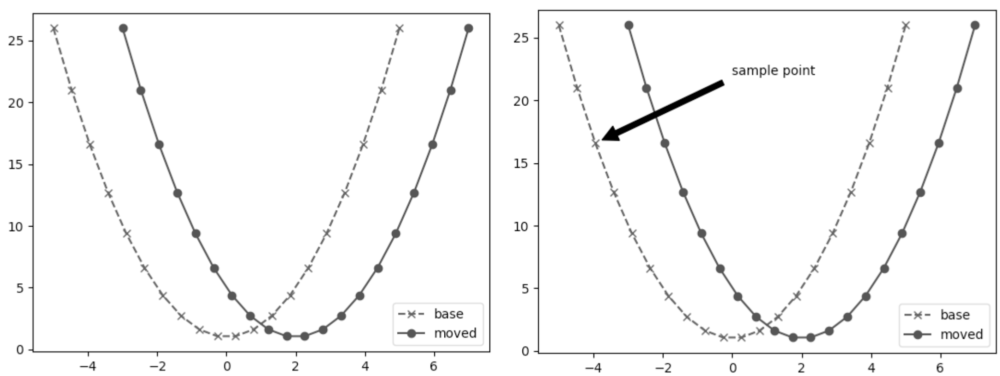


图 2-7　图例（左）与提示（右）

函数 annotate()的参数 arrowprops 是一个字典类型变量，其中可以输入的参数如表 2-3 所示。

表 2-3　arrowprops 参数常用内容

名称	解释
width	箭头宽度
frac	箭头的头部占整个箭头线的比例
headwidth	箭头头部的宽度
shrink	“箭头距离数据点和解释文本之间空隙”占“数据点至解释文本距离”的比例
facecolor	颜色

7. 添加标题

最后，可以通过 title()、xlabel()、ylabel()等函数设置绘图标题及坐标名称，比如：

```
>>> plt.title("Easy Matplot",                                         # 主标题
          fontdict = {"fontsize":20,}, loc='center')
>>> plt.xlabel('X values', fontdict = {"fontsize":12, })              # X 轴描述
>>> plt.ylabel('Y values', fontdict = {"fontsize":12, })              # Y 轴描述
```

以上代码执行效果读者可以自行尝试。Matlab 提供了很多进一步定制坐标系、刻度、图例、提示、各种箭头的支持函数与属性，这些细节超出了本书的机器学习应用目标，感兴趣的读者可以查阅官方网站关于 artist（Matplot 中所有可视元素的基类）、annotation、text、tick、patch、path 等类的详细文档。

2.2.2 子视图

在比较不同算法或数据集时经常需要一次绘制多个子图以便于对比，Matplot 提供了在一个绘图中建立多个子视图（subplot）/子坐标系的方式，并实现了灵活的子视图布局配置方法。

1. 用 subplot()布局

函数 subplot（numRows、numCols、plotNum）用于建立一个子坐标系，并用三个参数定义坐标系的大小与位置。可以想像该函数先将整个空间划分为 numRows×numCols 的网格，新坐标系的大小就是每个网格的空间大小，而 plotNum 参数用于指定坐标系位置在第几个网格。比如：

```
>>> ax1 = plt.subplot(2, 2, 1)          # plotNum 从 1 开始
>>> ax2 = plt.subplot(2, 2, 2)
>>> ax3 = plt.subplot(2, 2, 3)
>>> ax4 = plt.subplot(2, 2, 4)
```

以上代码建立四个子视图，效果如图 2-8（左）所示。其实还可以建立大小不相同、或有空隙的多个子视图，比如：

```
>>> ax1 = plt.subplot(2, 2, 3)          # 2×2 空间中的第三个网格（左下角网格）
>>> ax2 = plt.subplot(1, 2, 2)          # 1×2 空间中的第二个网格（右侧网格）
```

布局效果如图 2-8（右）所示。

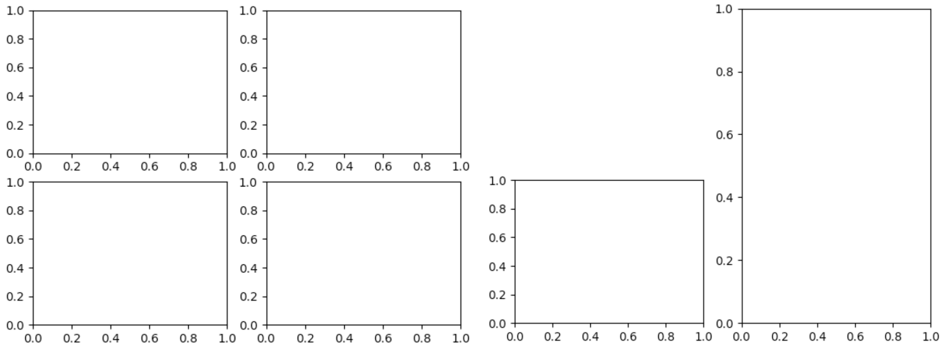


图 2-8　subplot()布局效果

2. 用 subplot2grid()布局

函数 subplot2grid()提供了一种更加灵活的布局方式，其通过四个参数定义子视图。

◎　Shape：网格空间的形状。

◎　loc：子视图在网格空间中的位置。

◎　rowspan：子视图跨越几个网格行，默认为 1。

◎　colspan：子视图跨越几个网格列，默认为 1。

举例如下，布局效果如图 2-9 所示。

```
>>> ax1 = plt.subplot2grid(shape=(3, 3), loc=(0, 0), colspan=2)
>>> ax2 = plt.subplot2grid(shape= (3, 3), loc= (1, 0), colspan=2)
>>> ax3 = plt.subplot2grid(shape= (3, 3), loc= (0, 2), rowspan=2)
>>> ax4 = plt.subplot2grid(shape= (3, 3), loc= (2, 0), colspan=3)
```

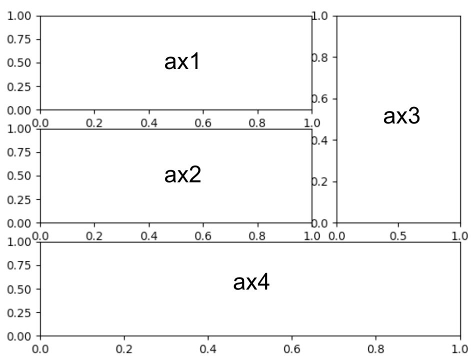


图 2-9　subplot2grid()布局效果

3. 子视图的绘制

开发者可以直接通过坐标系（axes）对象进行数据绘制，一些绘制函数名与原来调用 matplotlib.pyplot 包接口的函数名一致，比如 plot()、legend()等；而另外一些需要在原函数名前加“set_”前缀，比如 set_xticks()、set_yticks()等：尚未熟练掌握的开发者可在调用前查询 Matplot 在线文档。下面是一个在三个子图中分别绘制自变量取值在[0…2]时的典型幂函数、指数函数、对数函数的代码示例：

```
ax1 = plt.subplot(1, 3, 1)                          # 建立子视图
ax2 = plt.subplot(1, 3, 2)
ax3 = plt.subplot(1, 3, 3)

x = np.linspace(0, 2, 100)                          # 自变量数组

ax1.plot(x, x**0.5, label="y=power(x, 0.5)")        # 绘制数据点线图
ax1.plot(x, x**2, label="y=power(x, 2)")
ax2.plot(x, 0.5**x, label="y=power(0.5, x)")
ax2.plot(x, 2**x, label="y=power(2, x)")
ax3.plot(x, np.log2(x), label="y=log2(x)")
ax3.plot(x, np.log10(x), label="y=log10(x)")

ax1.legend()                                # 不设置 loc 参数，自动选择图例位置
ax2.legend()
ax3.legend()

plt.show()                                          # 显示
```

结果如图 2-10 所示。

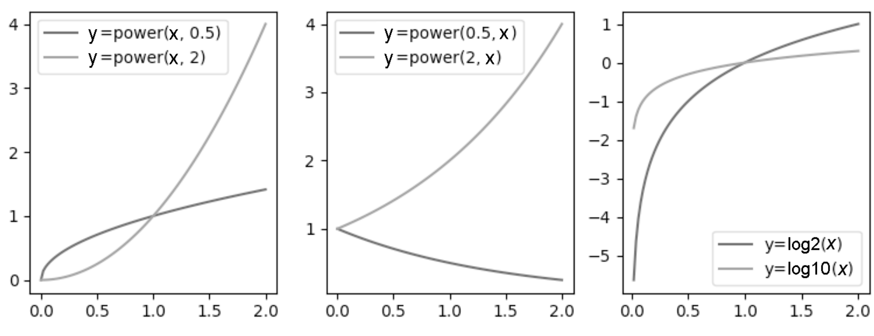


图 2-10　绘制常见初等函数

2.2.3　图像

在之前的学习中使用的都是用 plot()函数绘制点线图，从现在开始介绍其他绘制函数，本小节学习静态图像（image）绘制。

1. 图像数据

任何静态图像都可以看成一个 $M \times N$ 的像素矩阵，根据像素颜色的表达方式不同又可以进一步分为 $M \times N \times 3$（比如 RGB 像素，用三个数字表示颜色），$M \times N \times 4$（比如用 RGBA 四个数字表示颜色）等三维数组形式。在 Matplot 中用 image 库加载图像，比如：

```
>>>import matplotlib.image as mpimg            # 引入 image 库
>>>img = mpimg.imread('mantis.png')            # 用 imread()读入图像文件
>>>print(img)                                  # 打印图像内容
[[[ 0.          0.03137255  0.          1.        ] # 每个像素点由四个数值表达
  [ 0.03921569  0.04705882  0.02745098  1.        ]
  [ 0.01176471  0.02745098  0.07058824  1.        ]
  ...,
  [ 0.08235294  0.01176471  0.00392157  1.        ]
  [ 0.09019608  0.01960784  0.01176471  1.        ]
  [ 0.06666667  0.02352941  0.00784314  1.        ]]

 [[ 0.10196079  0.01960784  0.04705882  1.        ]
  [ 0.01176471  0.05098039  0.01960784  1.        ]
  [ 0.03921569  0.02745098  0.00784314  1.        ]
  ...,
]]
```

代码用 imread()函数读取图片文件后保存在变量 img 中，从打印的图像数据内容看，该 PNG 图像是一个三维数组，每个像素数据用四个数值分别保存 red、green、blue、alpha 颜色数值。

2. 显示图像

通过 imshow()函数即可将图像显示在坐标中，如图 2-11 所示。

```
>>>plt.imshow(img)
```

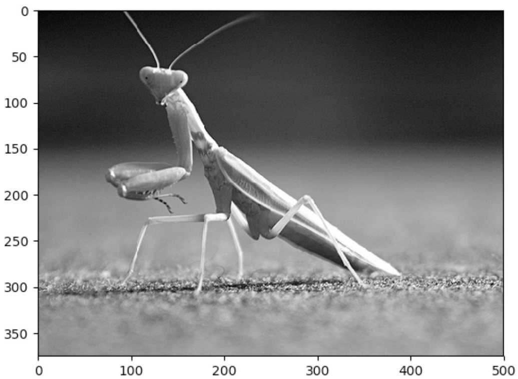


图 2-11　图像显示（取自深度学习图像库 ImageNet）

因为图像（图 2-11）左上角是矩阵（0, 0）位置点，因此显示图像的坐标系纵轴数值从零开始自上而下排列。

3. 色带（cmap）

在对图像颜色不敏感的应用场景，比如形状检测、手写识别等领域，通常不需要用三或四个数值进行真彩图像显示，取而代之的是只用一个数值表达颜色深浅。有计算机基础的读者可以知道这就是灰度图，但其实，在提供色带的情况下，灰度图也可用来显示彩色图像，色带可以在调用 imshow()时通过参数 cmap 传入。

以下代码可以从 RGB 图像中提取一个通道的颜色值成为灰度图像：

```
>>>grey_img = img[:, :, 0]                              # 提取第一个通道的颜色
>>>print(grey_img)
[[ 0.          0.03921569  0.01176471 ...,  0.08235294  0.09019608
   0.06666667]
 [ 0.10196079  0.01176471  0.03921569 ...,  0.07450981  0.09803922
   0.09019608]
 [ 0.08235294  0.07058824  0.05098039 ...,  0.10588235  0.09803922
   0.08627451]
 ...,
]
```

从打印的数据可知，灰度图像是一个 $M \times N$ 的二维数组/矩阵，每个像素只用一个数值表达颜色。那么这些数值都转变为什么颜色呢？这就要看 cmap 传入的色带是什么了。比如：

```
ax1.imshow(grey_img, cmap="binary")                     # 黑白灰度图
```

```
ax2.imshow(grey_img, cmap="winter")                    # 蓝绿冷色调
ax3.imshow(grey_img, cmap="Spectral")                  # 彩虹谱
```

其显示效果如图 2-12 所示，相同的二维图像由于 cmap 的不同而显示不同的颜色。

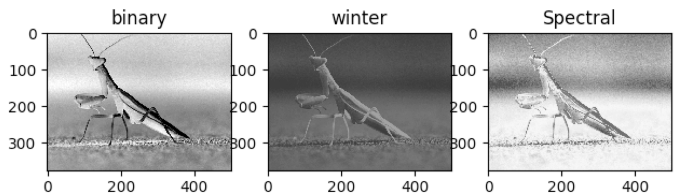


图 2-12　色带显示效果

代码中 cmap 参数传入的是色带名字，在 Matplot 中定义了若干色带名称，如图 2-13 所示是从 Matplot 官网截取的一组常用色带图例。

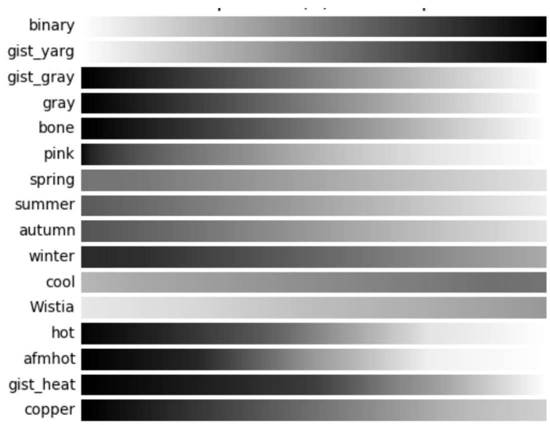


图 2-13　常用色带图例

4. 颜色直方图（histogram）

直方图是一种重要的图像处理工具，常用来进行色彩均衡化或增强对比度。在 Matplot 中用 hist()绘制图像直方图非常方便，比如：

```
>>>plt.hist(lum_img.ravel(), bins=256)
```

直方图本身是对各颜色像素点的数量统计，hist()中 bins 参数是直方图中横坐标上的粒度，该值越大统计得越精细。如图 2-14 所示是在不同 bins 值下的直方图显示效果。

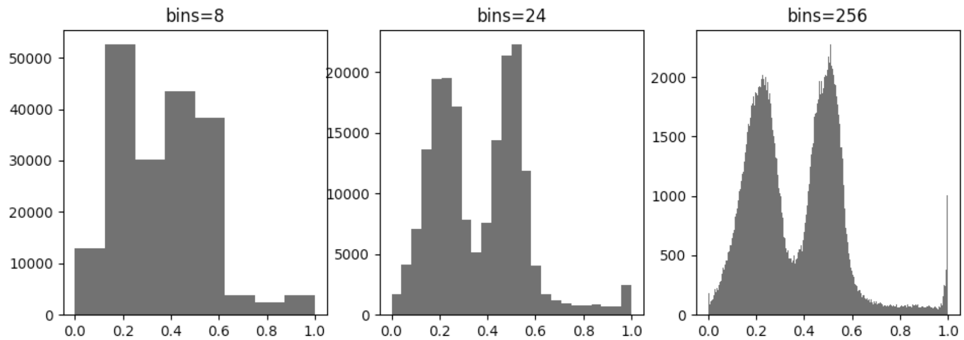


图 2-14　在不同 bins 值下的直方图显示效果

5. 插值（interpolation）

当小图片在电脑上放大显示时通常会有模糊感，原因就是当图像本身的像素点很少但要显示在像素点较多的画布上时，原有图像的像素无法铺满整个显示区域，导致计算机必须自己填充这些多余像素，这个填充的策略就是插值。在 Matplot 中集成了很多插值算法，在调用 imshow()时传入算法名称即可，比如：

```
thumb = mpimg.imread('mantis_thumb.png')   # 读入一个 64×64 像素的小图像

ax1.imshow(thumb, interpolation="nearest") # 给三个子图分别设置不同的插值算法
ax2.imshow(thumb, interpolation="bicubic")
ax3.imshow(thumb, interpolation="sinc")
```

三种插值算法的显示效果如图 2-15 所示。

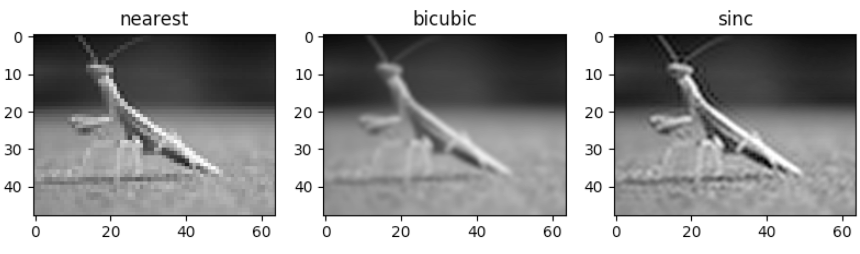


图 2-15　三种插值算法的显示效果

在 imshow()中可以设置的插值算法还包括 none、bilinear、spline16、spline36、hanning、hamming、hermite、kaiser、quadric、catrom、gaussian、bessel、mitchell、lanczos，读者可自行尝试。

2.2.4　等值图

等值图（contour）是一种用二维坐标系表达三维数据的绘图方式。普通的二维坐标的横坐标是自变量，纵坐标是因变量；等值图上的横坐标与纵坐标都是自变量，而因变量用图上的等值线表示。等值图一般用于不需要精确知道因变量数值，只需概览因变量变化层次的场景，比如地理海拔等高线。举例说明绘制方法，准备绘制数据如下：

```
x = np.linspace(-1, 2, 100)                # 自变量 x，长度为 100 的一维数组
y = np.linspace(-1, 2, 50)                 # 自变量 y，长度为 50 的一维数组
z = y.reshape((y.size, 1))*x               # 因变量 z，是一个 50×100 的矩阵
z = np.sin(z)                              # 计算因变量值
```

其中需要注意的是：因变量数组应该是一个二维数组/矩阵，这样才能包含所有自变量组合的情况。

思考：如果有一种图能绘制三、四、五……个自变量的情况，那么因变量就需要是一个三、四、五……维数组。

Matplot 中提供了两个等值图绘制函数，分别是 contour()和 contourf()，它们的使用方式完全一样，只是 contour()用等值线绘制，而 contourf()用实心区域绘制。调用方法举例：

```
cs = ax1.contour(x, y, z)                      # 绘制等值图
ax1.clabel(cs)                                 # 在等值线上标出 z 值
ax2.contourf(x, y, z)                          # 绘制实心区域图
```

绘制效果如图 2-16 所示，由于没有在 ax2 上调用 clabel()函数，所以右图没有显示 z 值。

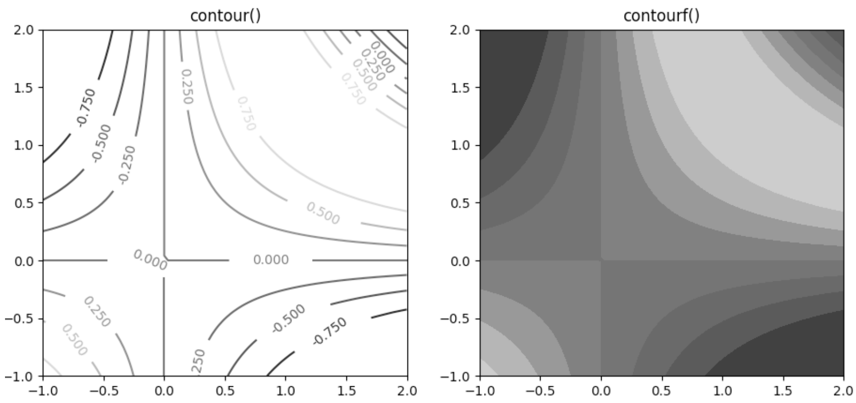


图 2-16　等值图绘制效果

2.2.5 三维绘图

当等值图无法满足三维数据的绘制场景时，可以借助三维工具包 mpl_toolkits.mplot3d 来绘制三维视图（plot3d）。引入该工具包后，在创建子视图时传入参数 projection=“3d”即可将该子视图转为三维坐标系：

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D    # 必须引入 mpl_toolkits.mplot3d
ax = plt.subplot(1, 1, 1, projection='3d') # 创建三维坐标系
```

此时创建的 ax 变量是一个类 Axes3D 的实例，该类提供了在三维坐标系上绘制各种 3D 图形的丰富接口。

1. 线

空间线是三维绘图中最简单的情况，绘制其只需要给出线段上的各点坐标，比如：

```
x = np.linspace(0, 10, 100)                        # X 轴
y = np.linspace(-10, 0, 100)                       # Y 轴
z = np.sin(x+y)                                    # Z 轴
ax.plot(x, y, z)
```

效果如图 2-17 所示。

2. 点

给出空间中的点，可以用 scatter()函数进行绘制，同时用 *c* 参数指定点的颜色，用 marker 定义点的形状，代码如下。

```
ax.scatter(x, y, np.sin(x+y), c="r", marker = ">")
ax.scatter(x, y, np.cos(x+y), c="b", marker = "<")
```

其中 marker 和 c 参数的取值同样可以参考学习基本绘图时的表 2-1 和表 2-2。以上代码效果如图 2-18 所示。

3. 曲面

通过 plot_surface()函数可以绘制出三维空间中的面，比如：

```
x = np.linspace(0, 10, 100)
y = np.linspace(-10, 0, 100)
x, y = np.meshgrid(x, y)                           # 将 x，y 变为二维数组
z = np.sin(x+y)                                    # 生成 z 值
```

```
    ax.plot_surface(x, y, z, cmap= "Spectral")      # 用色带指定的颜色绘制三维平面
```

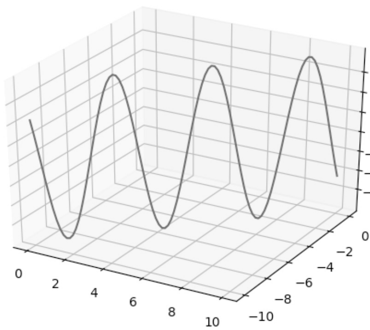


图 2-17　三维空间线效果

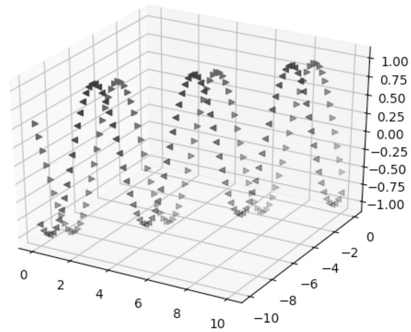


图 2-18　三维空间点效果

代码中 x、y、z 值都必须是二维数组，其中的值用于定义要绘制曲面上的点在 x、y、z 各自轴上的坐标，同时 z 中的数值除了用于定位坐标还用于结合 cmap 显示平面颜色，效果如图 2-19 所示。

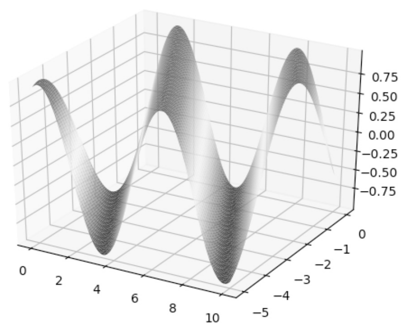


图 2-19　三维曲面效果

2.2.6　从官网学习

至此，读者已经学习了 Matplot 的基本技巧，不得不说这些内容对于整个 Matplot 库中的功能来说不够全面。但是有了这些实践作基础，一方面对本书后续的机器学习应用已经足够，另一方面读者已经有足够的知识阅读 Matplot 官方文档以学习其他特性。

Matplot 官网提供了足够多的代码实例演示繁多的绘图技巧，通常开发者可以直接通过下面的两个链接进行开发参考。

◎　实例集：https://matplotlib.org/2.1.0/gallery/index.html。

◎　API 参考：https://matplotlib.org/contents.html。

笔者最推崇的是官网精心准备的第一个链接中的实例集，其中提供了近 150 个代码片段演示不同的绘图技巧，如图 2-20 所示是其中一部分截图。

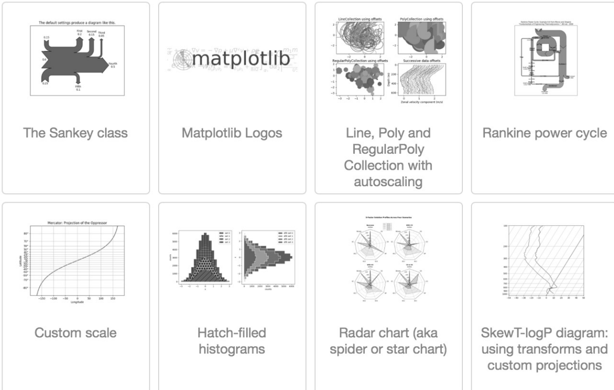


图 2-20　官网实例集部分截图

掌握直接从官网学习的能力的另一个好处是，能够体验绘图技巧的最新技术。Matplot 绘图工具包相对于本书中的其他数学或机器学习工具包来说功能更繁杂，并且对理论基础要求不高，对于机器学习开发者来说，在需要时随时参考、拿来即用是最佳策略。

2.3　Scipy

Scipy 是 Python 的开源算法库和数学工具包，几乎所有的 Python 机器学习工具包都使用了其中线性代数、微积分、插值、概率统计等数学模块。学习本节后读者可以自己调用这些函数库进行数学计算。在学习之前可用如下命令快速安装 Scipy：

```
#pip3 install scipy
#python3
>>> import scipy
>>> scipy.__version
1.1.0
```

目前为止 Scipy 的 pip 稳定版是 1.1.0 版本，也是本书学习与使用的版本。虽然在本书后续章节的案例中很少直接用到 Scipy，但通过学习 Scipy 可以用形象化的方法回顾中学和大学里的数学知识。

2.3.1　数学与物理常数

数学和物理中有一些常数在计算中经常用到，比如圆周率、重力加速度等。Scipy 为这些常数定义了常量，使用起来非常方便，比如：

```
>>> import scipy.constants as C                  # 引入常数包
>>> C.pi                                         # 圆周率
3.141592653589793
>>> C.g                                          # 引力常数
9.80665
```

如表 2-4 所示列出了 scipy.constants 包中定义的其他常用数学、物理与换算常数。

表 2-4　常用数学、物理、换算常数

类别	名称	值	含义
数学	golden	1.618033988749895	黄金分割比
物理	c	299792458.0	光速
	epsilon_0	8.854187817620389e–12	真空电容率
	h	6.62607004e–34	普朗克常数
	e	1.6021766208e–19	基本电荷
	R	8.3144598	普适气体常数
	alpha	0.0072973525664	精细结构常数
	N_A	6.022140857e+23	阿伏伽德罗常数
	k	1.38064852e–23	玻尔兹曼常数
	Rydberg	10973731.568508	里德伯常数
	m_e	9.10938356e–31	电子静止质量
	m_p	1.672621898e–27	质子静止质量
	m_n	1.674927471e–27	中子静止质量
	atm	101325.0	标准气压（帕斯卡）
换算	pound	0.45359236999999997	英磅/千克
	ounce	0.028349523124999998	盎司/千克
	degree	0.017453292519943295	角度/弧度
	inch	0.0254	英寸/米
	foot	0.30479999999999996	英尺/米
	yard	0.9143999999999999	码/米

续表

类别	名称	值	含义
换算	mile	1609.3439999999998	英里/米
	acre	4046.8564223999992	英亩/平方米
	gallon	0.0037854117839999997	加仑/立方米
	hp	745.6998715822701	马力/瓦特

2.3.2 特殊函数库

在 scipy.special 包中实现了众多特殊函数（special），其内容非常庞大，理解这些函数的具体算法涉及《复分析》《数学物理方法》《数值分析》等专业课程，这里只做简要介绍。special 包中主要的特殊函数如下。

◎ 艾里函数（airy）：即常微分方程 $\frac{d^2y}{dx^2}-xy=0$ 解的函数集。

◎ 椭圆函数（elliptic）：最初用于计算椭圆弧的长度，包括 Weierstrass's elliptic functions、Jacobi's elliptic functions、Jacobi's elliptic functions 等。

◎ 贝塞尔函数（bessel）：满足常微分方程 $x^2\frac{d^2y}{dx^2}+x\frac{dy}{dx}+(x^2-\alpha^2)y=0$ 的函数集，常用于柱面环境相关计算。

◎ 伽马函数（gamma）：即函数 $\Gamma(z)=\int_0^\infty x^{z-1}e^{-x}dx$，是分析学、概率论、偏微分方程等学科中的重要基础函数。

◎ 贝塔函数（beta）：即函数 $B(x,y)=\int_0^1 t^{x-1}(1-t)^{y-1}dt$，也可以用伽马函数定义为 $B(x,y)=\frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}$。

◎ 超几何函数（hypergeometric）：用超几何级数定义的函数，很多特殊函数都是它的特例或极限。

◎ 抛物柱面函数（par×olic cylinder）：满足微分方程 $\frac{d^2f}{dz^2}-(az^2+bz+c)=0$ 解的函数集。

◎　马丢函数（mathieu）：满足常微分方程 $\frac{d^2 f}{dz^2}-(a-2q\cos 2z)y=0$ 解的方程，其中 a 与 q 都是实数。

◎　开尔文函数（Kelvin）：是贝塞尔函数的 4 种特殊形式，应用于电磁学物理研究。

以上每类函数都用传统的数学公式中的符号名称定义了若干函数方法用于计算函数值或按照函数生成分布序列，比如对于艾里函数，special 包中的可调用函数包括 airy()、airye()、ai_zeros()、bi_zeros()、itairy()等。这里以贝塞尔函数之一的 jv()演示其 special 包的使用方法。对其他函数的调用可查阅在线文档。

jv(v, z)函数是对贝塞尔函数中的如下公式的实现：

$$J_\alpha(x)=\sum_{m=0}^{\infty}\frac{(-1)^m}{m!\Gamma(m+\alpha+1)}\left(\frac{x}{2}\right)^{2m+\alpha}$$

自行编写该函数实现比较麻烦，除了实现公式目标还要考虑性能优化，而使用 Scipy 可以直接调用相应函数：

```
import scipy.special as spl
x = np.linspace(0,20,500)                              # 500 个采样点的 x 值
for i in range(3):
    y = spl.jv(i, x)
    plt.plot(x, y, '-', label="J%d"%i)
```

以上代码绘制 α 为三种不同值时的贝塞尔函数曲线，效果如图 2-21 所示。

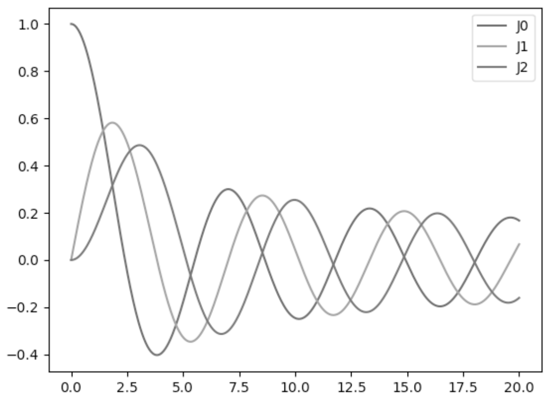


图 2-21　贝塞尔函数曲线

2.3.3 积分

Scipy 的积分（integrate）包 scipy.integrate 中主要实现了三类功能：定积分计算、数值采样积分计算、解常微分方程。通过举例理解积分计算功能：假设有函数 $y=x^2+3$，该函数是如图 2-22 所示的抛物线。

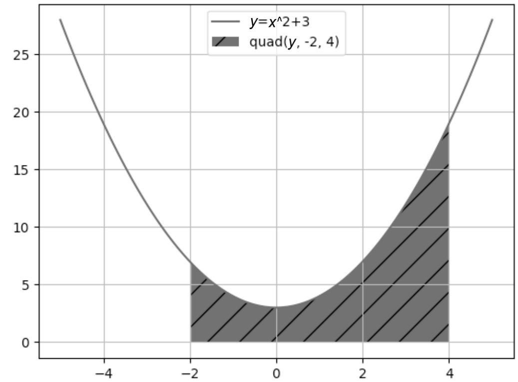


图 2-22 函数 $y=x^2+3$

定积分 $\int_{-2}^{4} x^2+3\mathrm{d}x$ 在平面坐标中的物理意义就是图中阴影部分所占的面积，而 integrate 包就提供了若干个这样的求定积分函数，本例的定积分可通过 quad()函数求出：

```
>>>from scipy import integrate
>>>Y = lambda x: x**2+3                    # 定义被积分函数
>>>print(integrate.quad(Y, -2, 4))
(42.0, 4.662936703425657e-13)
```

代码中给 quad()函数传递了三个参数，第一个参数就是被积分函数，后两个参数分别是定积分的下界和上界。函数有两个返回值，第一个返回值就是定积分计算结果，也就是图 2-22 中阴影部分的面积；第二个返回值是对计算误差的估计，本例中是一个非常小的值。与 quad()函数使用方法类似的还有 dblquad()、tplquad()等，分别用于计算二重、三重积分。

在 integrate 包中还提供了数值采样积分计算函数，所谓的数值积分是一种对定积分的近似计算方法。数值积分的输入不是一个函数，而是该函数的若干采样点。仍用图 2-22 中的函数，通过数值积分函数进行计算：

```
>>>Y = lambda x: x**2+3                    # 定义函数，用于采样
>>>x = np.linspace(-2,4,10)                # 在[-2, 4]内的 10 个 x 采样点
```

```
>>>y = Y(x)                              # 计算采样点的 y 值
>>>print(integrate.trapz(y, x))          # 使用 trapezoidal rule 计算数值积分
42.4444444444

>>>x = np.linspace(-2,4,50)              # 50 个采样点
>>>y = Y(x)
>>>print(integrate.trapz(y, x))
42.0149937526

>>>x = np.linspace(-2,4,100)             # 100 个采样点
y = Y(x)
print(integrate.trapz(y, x))
42.0036730946
```

注意：在调用数值积分函数时，只以采样点坐标作为输入参数，而没有用被积分函数作为输入。从三次计算可知，采样点越多所计算的积分结果越接近用 quad()计算的真实定积分结果。另外，数值积分相对于普通定积分计算的另一个优势是计算速度较快。

在数学上数值采样积分有多种计算方法，所以在 integrate 包中除了 trapz()函数，还提供了 simps()、romb()等其他数值采样积分计算函数。

2.3.4　优化

Scipy 的优化（optimize）包 scipy.optimize 主要实现了三类功能：求函数最小值点、线性和非线性拟合、方程组求解。每种问题都有若干函数适用于不同场景，本节将举例学习。

1. 求函数最小值点

假设有多元函数 $y = x_0^3 + x_1^3 + \cos(x_2)$，想求当 x_0、x_1、x_2 为何值时 y 获得最小值。为该问题编码如下：

```
>>>from scipy.optimize                           # 引用工具包

>>>func = lambda x: x[0]**3+x[1]**3 + np.cos(x[2]+1)   # 定义被求最小值函数
>>>x0 = np.array([0, 0, 0])                      # 猜测的结果
>>>res = optimize.minimize(func, x0)             # 调用 minimize()函数求最小值点

>>>print("y=%f when x=%s"%(func(res.x), res.x))        # 用 res.x 读取结果
y=-1.000000 when x=[ 0.         0.         2.14159739]
```

求最小值函数 minimize()有两个必要参数：函数对象 func、初始猜测 x0（initial guess）。func 的参数是一个数组，其中的每个元素是该函数的一个变量；x_0 有两个作用，一是告诉 minimize()函数 func 是几元变量函数，二是给出一个起始搜索点。在本例中，函数 $y(x_0, x_1, x_2)$在点（0, 0, 2.14159739）处达到最小值，此时 y=–1.0。

2. 拟合

所谓拟合是指这样一种问题：已知某组数据是由一个形式已知的函数产生而得，需要知道该函数的若干固定参数值；并且，允许这组已知数据中有随机噪声存在。下面是官网上的一个应用举例：假设有经函数 $y=ae^{-bx}+c$ 产生的若干有噪声数据，求 a、b、c 的值。

```
>>>from scipy.optimize import curve_fit

>>>def func(x, a, b, c):                              # 定义函数 y = a×e^(-bx) + c
      return a * np.exp(-b * x) + c

# 如下几行用 a=2.5, b=1.3, c=0.5 生成带噪声的一组拟合数据
>>>xdata = np.linspace(0, 4, 50)                      # 生成拟合数据的 x 轴值
>>>y = func(xdata, 2.5, 1.3, 0.5)                     # 生成拟合数据的 y 轴值
>>>np.random.seed(1729)
>>>y_noise = 0.2 * np.random.normal(size=xdata.size)   # 生成噪声
>>>ydata = y + y_noise                                # 在 y 轴上加入噪声

>>>popt, pcov = curve_fit(func, xdata, ydata)  # 调用 curve_fit()拟合
>>>print(popt)                                        # 打印拟合得到的 a，b，c 参数
[ 2.55423706  1.35190947  0.47450618]
```

最后得到的拟合参数是 a=2.55423706、b=1.35190947、c=0.47450618，与生成该组数据的原参数 a=2.5、b=1.3、c=0.5 相近，不能完全匹配的原因是拟合数据中存在噪声。

3. 方程组求解

optimize 包中的 root()函数提供了普通方程组求解的功能。假设有如下齐次方程组：

$$\begin{cases} x^2+y^2-\dfrac{z}{5}-3=0 \\ x^2+\dfrac{y}{5}-z+1=0 \\ x+y+z-7=0 \end{cases}$$

求解其中 x、y、z 值的代码如下：

```
>>>from scipy import optimize

>>>def fun(x):                                        # 在一个函数中定义方程组
     return [x[0]**2 + x[1]**2 -x[2]/3 - 3,
          x[0]**2 +x[1]/5- x[2] +1,
          x[0]+x[1]+x[2]-7
     ]

>>>sol = optimize.root(fun, [0, 0, 0])          # 用 root()函数求解
>>>print(sol.x)                                       # 打印结果
[ 1.68344818  1.23546169  4.08109013]
```

与 minimize()函数类似，root()函数也要求有两个必要参数：函数对象与初始猜测。本例中的求解结果为 x=1.68344818，y=1.23546169，z=4.08109013，读者可自行演算。

2.3.5　插值

插值（interpolation）是一种根据已有数据生成新数据的方法，新数据服从于与已有数据相同的算法或分布。从这个角度看其与 optimize 包中拟合问题适用场景有相似之处，其区别在于：

◎　拟合允许有噪声的存在，而插值在计算时认为已有数据是绝对正确的。

◎　使用拟合需要先知道函数/分布形式，而插值只需要有样本数据即可。

Scipy 中的插值函数可分为一维插值、多维插值、样条插值等几种类型。以一维插值举例说明：

```
from scipy import interpolate                        # 引入 interpolation 包
import scipy.constants as C

x = np.linspace(0, C.pi*2, num=10, endpoint=True)  # 生成样本数据 x 值
y = np.sin(x)                                     # 按照 sin 曲线生成样本数据 y 值

interp_line = interpolate.interp1d(x, y)        # 生成普通线性插值函数
interp_cubic = interpolate.interp1d(x, y, kind='cubic')# 生成 cubic 插值函数

xnew = np.linspace(0, C.pi*2, num=33, endpoint=True)   # 生成插值数据 x 值
ynew_line = interp_line(xnew)                     # 用线性插值函数计算 y 值
```

```
ynew_cubic = interp_cubic(xnew)                 # 用 cubic 插值函数计算 y 值

# 绘制三种数据：样本数据、线性插值结果数据、cubic 插值结果数据
plt.plot(x, y, 'o', xnew, ynew_line, '-', xnew, ynew_cubic, '--')
plt.legend(['data', 'linear', 'cubic'], loc='best')
plt.show()
```

如代码中所示，使用一维插值生成新数据时，先用 interp1d()生成插值函数，然后用生成的函数计算新数据的 y 值。本段代码用两种类型的插值算法生成插值函数，分别是默认使用的线性算法和 cubic 算法，代码结果如图 2-23 所示。

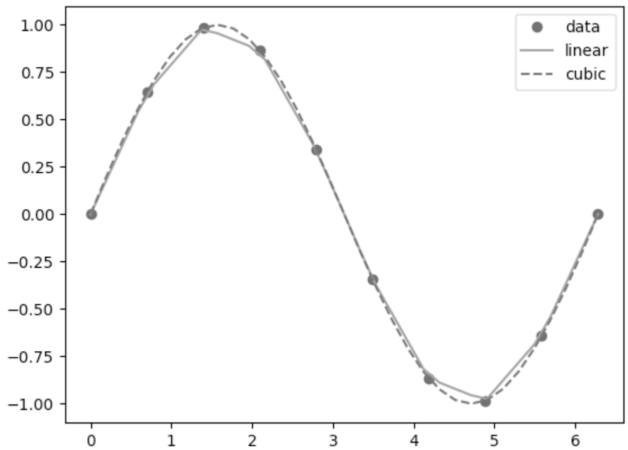


图 2-23　用线性算法和 cubic 算法生成插值函数的结果

图 2-23 中的 10 个原始数据用圆点表示，线性插值生成的数据用实线表示，cubic 插值生成的数据用虚线表示。本例显示了插值算法的极大魅力：通过一个正弦周期的 10 个均匀分布采样点就可以在极大程度上还原原始信号。

在 interpolation 包中还有其他函数应用于不同的插值环境，比如同样应用于一维变量的 BarycentricInterpolator()、PchipInterpolator()，用于多维变量的 interpn()、Rbf()、NearestNDInterpolator()、RegularGridInterpolator()，样条插值 BSpline()、BivariateSpline()、splrep()等。

2.3.6　离散傅里叶

傅里叶变换（Fourier Transform）是一种可以将任何连续信号转换为无限多个周期性

函数相加形式信号的转换方法。其典型的数学公式是：

$$y(\xi)=\int_{-\infty}^{\infty} f(x)\mathrm{e}^{-2\pi \mathrm{i} x\xi}\mathrm{d}x\text{，}\ \xi\text{是任意实数}$$

上述公式积分结果组成的连续区间是周期性函数。而计算机领域的数字信号特性关注的主要是离散傅里叶变换（Discrete Fourier Transform），其变换方式从积分变为求和，积分/求和区间由无穷变为有限。在离散傅里叶变换中，原始信号是一个数字序列，变换后的傅里叶信号是另一个数字序列。变换后的数字序列越长，则越能表征原始信号。假设原始信号是长度为 N 的数字序列（$x[0], x[1], x[2], \ldots, x[N-1]$），其快速傅里叶变换（Fast Fourier Transform）数学公式是：

$$y[k]=\sum_{n=0}^{N-1}\mathrm{e}^{-2\pi\mathrm{i}\frac{kn}{N}}x[n]$$

用 k=0, 1, 2…K 逐个执行该公式，就可以得到变换后的傅里叶数字序列（$y[0], y[1], y[2], \ldots, y[K-1]$）。很容易作该公式的反函数，将变换后的序列还原为原始信号数字序列：

$$x[n]=\frac{1}{N}\sum_{k=0}^{N-1}\mathrm{e}^{-2\pi i\frac{kn}{N}}y[k]$$

在 scipy.fftpack 包中就提供了若干个这样的正向与反向傅里叶变换函数。快速傅里叶变换举例：

```
>>>from scipy.fftpack import fft, ifft     # 引入快速傅里叶变换函数与反函数

>>>x = np.array([2.0, 3.0, -1.0, -3.0, 0.5])   # 原始信号
>>>y = fft(x)                                  # 快速傅里叶变换
>>>print(y)                                    # 打印出变换后的序列
[ 1.50000000+0.j        6.31762746-3.5532118j -2.06762746+0.4326499j
 -2.06762746-0.4326499j  6.31762746+3.5532118j]

>>>yinv = ifft(y)                              # 反傅里叶变换
>>>print(yinv)                                 # 打印出还原的信号
[ 2.0+0.j  3.0+0.j -1.0+0.j -3.0+0.j  0.5+0.j]
```

注意：诸如 2.0+0.j 等数值是 Python 中复数（complex）的表达方式。

本例先是用 fft()将原始信号变换为傅里叶序列，然后通过 ifft()反变换还原出原始信号序列，其还原的结果与原始信号完全相同。根据变换公式的不同，fftpack 中提供了快速变

换、正弦变换、余弦变换等几种不同的傅里叶变换函数，不再一一举例。

2.3.7 卷积

信号处理包 scipy.signal 又是一个非常庞大的函数库，其领域包含了卷积计算、B 样条变换、小波变换、光谱分析等专业课题，多数应用在图形图像处理领域。由于卷积（converlution）与机器学习的特殊关系——众所周知卷积神经网络是深度学习的重要实践方法，本节借卷积计算解释 signal 包的开发使用方法。

注意：如前文所述，在 Scipy 中信号就是一个数组，该数组可以是一维或多维。

在数学泛函分析中，卷积是通过两个函数 f 和 g 生成第三个函数的一种数学算子；相应地在信号处理中，其就是将两个信号结合生成第三种信号的过滤算法。设有长度分别为 K、M 的一维信号 x 和 h，则它们的卷积结果长度为 N=K+M−1，结果信号中每个数值的计算方法是：

$$y[n]=\sum_{k=\max(n-M,0)}^{\min(n,k)} x[k]h[n-k]$$

根据这个公式，如果将 x 信号视为被过滤信号，h 视为过滤器；则卷积 y 就是将 h 在 x 上从头到尾处理（相乘）一次的结果。

调用 signal 包中的 convolve()函数即可完成该卷积运算，比如：

```
>>>from scipy import signal                    # 引入信号处理包

>>>x = np.array([1.0, 2.5, 3.0, 2.0])          # 定义被卷积信号
>>>h = np.array([0.7, 1.3])                    # 卷积信号

>>>print(signal.convolve(x, h))                # 调用 convolve()计算并打印结果
[ 0.7   3.05  5.35  5.3   2.6 ]
```

读者可以根据公式自行演算结果。卷积还可以在一维以上的信号上进行计算，比如：

```
>>>x = np.array([[1.0, 2.5, 3.0, 2.0], [2.0, -0.3, 9.1, 5.8], [3.7, 2.5, 2.0, 4.2]])
>>>h = np.array([[0.7, 1.3], [4.7, 5.0]])

>>>print(signal.convolve(x, h))
[[  0.7    3.05   5.35   5.3    2.6 ]
```

```
 [  6.1   19.14  32.58  40.29  17.54]
 [ 11.99  15.15  45.92  78.3   34.46]
 [ 17.39  30.25  21.9   29.74  21.  ]]
```

上述代码中的被卷积信号 x 是 3×4 的矩阵，卷积信号 h 是 2×2 的矩阵，卷积结果 y 是 4×5 的矩阵；y 在两个维度上的长度仍然符合 $N=K+M-1$ 的规则。

2.3.8　线性分析

在机器学习 Python 软件包中的向量、矩阵等线性空间使用 Numpy 数组表达。对于在线性空间上最简单的加、减、乘、除等运算直接使用相应运算符就可调用；而对于更复杂的运算，可以使用 scipy.linalg 包中提供的功能，其中包括矩阵转置（inverse）、求特征值（eigenvalue）、各种分解计算（奇异值分解、QR 分解、QZ 分解、矩阵三角化）等功能。

1. 基本运算

基本的加、减、乘、除可以直接调用运算符：

```
>>>a = np.array([[1,2], [3, 4]])                     # 用 Numpy 定义矩阵
>>>b = np.array([[-1, 0], [1, -2]])

>>>c = a * b                                         # 矩阵相乘
>>>d = a - b                                         # 矩阵相减
>>>e = 3 * a                                         # 标量与矩阵相乘
```

如需线性转置，可调用 inv()函数，比如：

```
>>>from scipy import linalg                          # 引入 linalg 包
>>>f = linalg.inv(a)                                 # 调用 inv()转置
>>>print(f)
[[-2.   1. ]
 [ 1.5 -0.5]]
```

2. 特征值与特征向量

在线性代数中，将一个矩阵 $\boldsymbol{A}$ 乘以一个向量 $\boldsymbol{v}$ 得到的结果可以看成对该向量进行的一次线性变换。如果该变换不改变向量 $\boldsymbol{v}$ 在线性空间中的方向而只改变 $\boldsymbol{v}$ 的长度，此时必有标量 λ 满足等式：

$$\boldsymbol{A}\boldsymbol{v} = \lambda \boldsymbol{v}$$

此时称 λ 是 $\boldsymbol{A}$ 的特征值，$\boldsymbol{v}$ 是 $\boldsymbol{A}$ 的特征向量。一个矩阵可以有多个特征值与特征向量，通过 linalg 中的函数 eig()即可计算得出它们：

```
>>>A = np.array([[1, 2], [3, 4]])                  # 定义矩阵
>>>la, v = linalg.eig(A)                           # eig()返回特征值与特征向量

>>>print("the eigenvalues is: ", la)               # 特征值
the eigenvalues is:  [-0.37228132+0.j  5.37228132+0.j]

>>>print("the eigenvectors is: ", v)               # 特征向量
the eigenvectors is:  [[-0.82456484 -0.41597356]
 [ 0.56576746 -0.90937671]]
```

本例中计算了矩阵$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$的两个特征值和特征向量。

3. 奇异值分解

奇异值（singular value）分解在机器学习的数据压缩、降维等领域有广泛应用，其特点是能够将任意矩阵 $\boldsymbol{A}$ 转换为用奇异值对角阵与两组正交基表达的方式：

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^h$$

其中 $\boldsymbol{U}$ 是左奇异向量组成的正交基矩阵；$\boldsymbol{\Sigma}$ 是对角阵，对角线上的就是奇异值；$\boldsymbol{V}$ 是右奇异向量组成的正交基矩阵；$\boldsymbol{V}^h$ 是矩阵 $\boldsymbol{V}$ 的共轭转置。

调用 linalg 包中的函数即可计算上述公式中的奇异值分解矩阵，比如：

```
>>>A = np.array([[1,2,3],[-1,-2,-3]])                  # 原矩阵
>>>m, n = A.shape
>>>U,s,Vh = linalg.svd(A)                              # 奇异值分解
>>>Sig = linalg.diagsvd(s,m,n)                         # 生成奇异值对脚阵

>>>print("Matrix U: ", U)                              # 左正交基矩阵
Matrix U:  [[-0.70710678  0.70710678]
         [ 0.70710678  0.70710678]]

>>>print("Matrix Sigma: ", Sig)                        # 奇异值矩阵
Matrix Sigma:  [[  5.29150262e+00   0.00000000e+00   0.00000000e+00]
[  0.00000000e+00   3.77456466e-16   0.00000000e+00]]
```

```
>>>print("Matrix Vt: ", Vh)                          # 右正交基矩阵的共轭转置
Matrix Vt:  [[ -2.67261242e-01  -5.34522484e-01  -8.01783726e-01]
            [ -9.63624112e-01   1.48249863e-01   2.22374795e-01]
            [ -2.03325294e-16  -8.32050294e-01   5.54700196e-01]]
```

任何一个 $m\times n$ 矩阵 $\boldsymbol{A}$ 的奇异值个数等于 $\min(m,n)$，上述代码中计算得到的矩阵 $\begin{bmatrix} 1 & 2 & 3 \\ -1 & -2 & -3 \end{bmatrix}$ 的两个奇异值是 5.29150262e+00 和 3.77456466e-16。

在奇异值应用中，公式 $\boldsymbol{A}=\boldsymbol{U\Sigma V}^h$ 可以用奇异值与奇异向量乘积和的方式表达：

$$\boldsymbol{A}=\sigma_1(\mu_1 \quad 0 \quad 0 \quad \cdots)\begin{pmatrix} \upsilon_1^h \\ 0 \\ 0 \\ \vdots \end{pmatrix}+\sigma_2(0 \quad \mu_2 \quad 0 \quad \cdots)\begin{pmatrix} 0 \\ \upsilon_2^h \\ 0 \\ \vdots \end{pmatrix}+\sigma_3(0 \quad 0 \quad \mu_3 \quad \cdots)\begin{pmatrix} 0 \\ 0 \\ \upsilon_3^h \\ \vdots \end{pmatrix}+\cdots$$

其中 $\sigma_1,\sigma_2,\sigma_3\ldots$ 是由大到小排列的奇异值标量，$\mu_1,\mu_2,\mu_3\ldots$ 是相应的左奇异向量，$\upsilon_1^h,\upsilon_2^h,\upsilon_3^h\ldots$ 是右奇异值向量共轭转置。而所谓的奇异值有损压缩、数据降维就是只取该表达式的前面若干个奇异值较大的分量用来表征矩阵 $\boldsymbol{A}$；达到的效果是大大减少了数据量，同时保留了 $\boldsymbol{A}$ 中的主要信息。在第 5 章将详细讨论这一重要的机器学习应用场景。

2.3.9　概率统计

在 scipy.stats 包中主要提供了各种随机分布的样本生成与统计函数，这些分布在本书中的大部分机器学习算法原理中都有应用，本节举例说明 stats 包中各分布的使用方法，各种分布的具体内容将穿插在后续的各个章节中随着相应算法一起学习。

1. PDF 与 CDF

在本书或其他机器学习书籍/文档中会反复用到的两个概念是分布的概率密度函数 PDF（Probability Density Function）与累积概率分布函数 CDF（Cumulative Distribution Function)。任何事件发生的概率应该在 0～1 之间，PDF 就是用来衡量某个值或某个区间事件发生可能性大小的函数，而 CDF 是从$-\infty$到某个值的区间内事件发生可能的大小。因此，CDF 一定是一个非严格递增函数，其值最终达到 1。

任何一个分布既可以用 PDF 描述，也可以用 CDF 描述，两者本质并无差别，只是根

据场景不同而适时被使用。如图 2-24 所示是同一个连续分布 gamma(*a*=2)的 PDF 和 CDF。

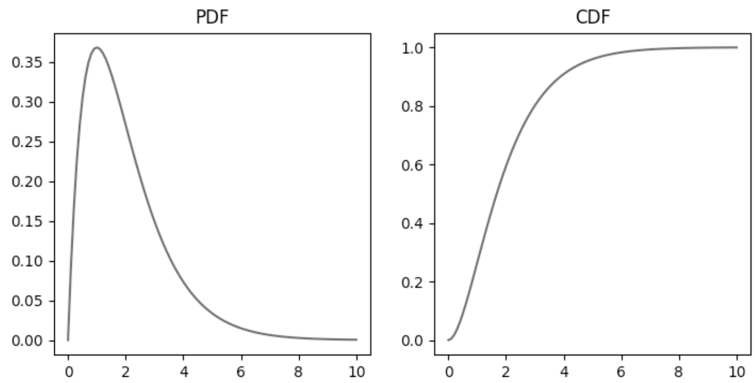


图 2-24　同一个连续分布 gamma(a=2)的 PDF 和 CDF

本例中，由于分布本身在定义域 2 附近有较高概率，因此 PDF 在该区域内值较高，而相应的 CDF 在该区域内上升很陡峭；因为分布在大于 6 以后的出现概率非常小，所以 PDF 在该区域内取值趋于零，而 CDF 在该区域内仍然保持上升趋势，只是幅度已经很平缓。

2. 常用分布

stats 包中的随机分布由三大类组成：连续分布、离散分布、多元分布。表 2-5 列出了本书中常用的分布。

表 2-5　常用分布

类型	在 scipy.stats 中的名字	说明
连续分布	beta	B 分布，定义域范围在[0, 1]之间
	cosine	余弦分布，即三角函数中的 cos()分布
	expon	指数分布，用于表示独立随机事件发生的时间间隔
	gamma	伽马分布，符合伽马函数 $\Gamma()$ 的分布
	laplace	拉普拉斯分布，两个背靠背指数分布组成的尖顶分布
	norm	正态分布、高斯分布
离散分布	bernoulli	伯努利分布，只包括两个值 0、1 的概率分布
	binom	二项分布，*n* 次独立伯努利分布试验后结果为 1 的次数的分布
	logser	对数离散分布，满足麦克拉伦数列的对数分布

续表

类型	在 scipy.stats 中的名字	说明
离散分布	poisson	泊松分布，单位时间内随机事件发生次数的概率分布
多元分布	multivariate_normal	多元正态分布
	dirichlet	狄利克雷分布，是 B 分布的多元版本
	multinomial	多项式分布，是二项式分布的多元版本

各种分布都对应了固定的数学公式，这里不再列举。本书在用到它们时会通过对应的 PDF 和 CDF 进行形象化的讲解。

3. 分布上的常用方法

在 stats 包中所有分布类型对象都有下列方法进行基于分布的计算。

◎ rvs()：生成随机序列。

◎ pdf()：计算概率密度函数。

◎ cdf()：计算累积概率分布函数。

◎ sf()：残存函数，即 1-cdf()。

◎ ppf()：百分比函数，是 cdf()的反函数。

◎ isf()：反残存函数，即 sf()的反函数。

◎ stats()：获取分布的均值、方差、偏度（Skewness）、峰度（Kurtosis）。

◎ fit()：输入样本，拟合计算分布参数。

以 gamma 分布举例它们的调用方式：

```
>>>from scipy.stats import gamma              # 引入 gamma 分布

>>>a=2                                        # 定义 gamma 分布的形状参数 a

>>>gamma.rvs(a, size=5)                       # 在 gamma 上生成 5 个随机值
[ 1.48897709  0.70248658  4.01956535  1.84392153  0.89747732]

>>>gamma.pdf(np.linspace(0, 9, 10), a=a)    # 在区间 0~9 上计算 10 个概率密度
[ 0.         0.36787944  0.27067057  0.14936121  0.07326256  0.03368973
```

```
  0.01487251  0.00638317  0.0026837   0.00111069]

>>>gamma.stats(a, moments='mvsk')              # 计算均值、方差、偏度、峰度
(array(2.0), array(2.0), array(1.414213562373095), array(3.0))
```

在上述代码中，a 是 gamma 分布必需的形状参数；由 pdf()结果可知该分布在值 1～3 附近出现可能性较大；从 stats()得知分布均值与方差均为 2.0，偏度为 1.414213562373095，峰度是 3.0。

stats()函数的 moments 参数用于指出需要返回哪些数值，没有学过统计学的读者可能对偏度和峰度的概念不熟悉，其实可以从字面意思理解：偏度代表的是 PDF 左偏（偏度> 0）或右偏（偏度< 0）；峰度表示的是极端差值对总体方差的贡献度，一般来说峰值越高分布的 PDF 越尖锐或者越低洼。

4. 核密度估计

以上列举的函数都是在已知分布类型的情况下进行样本生成或分布计算；而在概率统计中其逆向运算也非常实用，即在不知道分布类型的情况下从已有样本数据产生分布的 PDF。核密度估计（Kernel Density Estimation，KDE）就是一种完成这项任务的方法。

在 stats 包中提供了 gaussian_kde()函数进行高斯核密度估计，其用法举例如下：

```
>>>from scipy.stats import gaussian_kde                  # 引入 gaussian_kde

>>>x = np.array([0, 1, 1.5 , 2, 4, 6.5, 7, 7, 7.9, 8, 9, 10]) # 样本数据

>>>kde = gaussian_kde(x)                                 # 用样本数据估计分布
>>>print(kde)
<scipy.stats.kde.gaussian_kde object at 0x11571a9b0>   # 查看核对象类型

>>>plt.plot(x, np.zeros(x.shape), 'b+', ms=20)          # 绘制样本数据

>>>x_eval = np.linspace(0, 10, num=200)        # 生成评估数据
>>>y_eval = kde(x_eval)                        # 用 kde 计算评估数据的 PDF
>>>plt.plot(x_eval, y_eval, '-')               # 绘制评估数据

plt.show()
```

以上代码通过在 gaussian_kde 对象初始化时传入原始样本数据来完成分布估计，其后给该对象传入评估数据时，可以给出相应的 PDF，效果如图 2-25 所示。

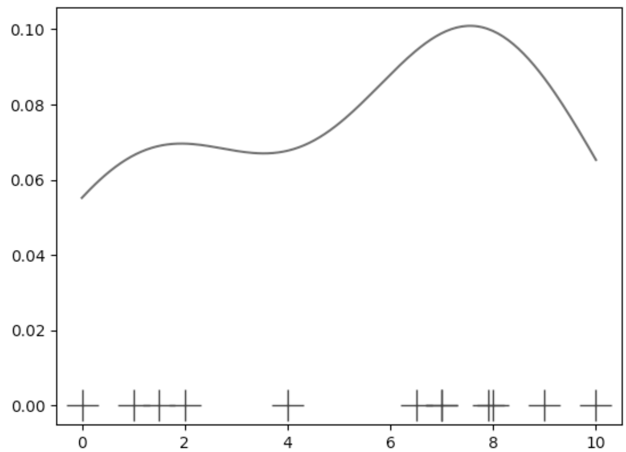


图 2-25　高斯核估计 PDF 效果

图 2-25 中底部的一排“加号+”是原始的样本数据，而上面的曲线就是用核方法估计出的 PDF。这样，通过核方法就可以完成从任意原始样本集评估其他样本发生可能性的任务。

思考：Numpy、Scipy、Matplot 分别具有什么样的功能？

2.4　本章内容回顾

◎　Numpy、Scipy 是基础数据科学包，几乎任何其他 Python 机器学习工具都在它们的基础之上开发。Matplot 用于形象化展示数据，也是数据研究开发过程的必备工具。

◎　Numpy 和 Scipy 底层用 C/C++开发，所以提供了高效数据处理能力。

◎　Numpy 提供了围绕多维数组 ndarray 开发的一系列基本数据操作：构造、访问、增、删、改等。

◎　轴是 Numpy 中很多函数的参数，其意义是“沿着 ndarray 的第几个维度执行”。

◎　Numpy 中的全函数（universal functions）提供了基础的数值运算函数。

◎　广播机制为不同形状的 ndarray 之间进行运算提供了可能性，但需遵守其四条广播规则。

- ◎ 运用 Matplot 绘制基本的点线图、图像、等值图、三维图等，用 title、legend、annotation 等进行装饰。
- ◎ 用 Scipy 进行特殊函数计算、积分计算、拟合、解方程组、插值计算。
- ◎ 理解离散傅里叶分析的原理和用 scipy.fftpack 计算的方法。
- ◎ 理解卷积原理，掌握 scipy.signal 中的调用方法。
- ◎ 应用 scipy.linalg 进行矩阵特征值分析、奇异值分解。
- ◎ 概率的两种表达方法：PDF 和 CDF。
- ◎ 应用 scipy.stats 中的各种概率分布模型。
- ◎ 核密度估计方法用于解决从已有样本中生成分布 PDF 的问题。

3

第 3 章 有监督学习：分类与回归

完成了前面一章的基础工具学习后，可以开启真正的机器学习之旅了。本章学习有监督学习的两大类问题：分类与回归。多数旧的文献把它们分开来介绍，但是由于当前大多数的有监督学习模型都可以同时应用于这两类问题，本书将它们合并在了一起。本章学习的主要内容如下。

◎ 线性回归：最适合入门的模型，包含 OLS、Ridge、Lasso。

◎ 随机梯度下降：一类最基本的优化算法，是理解神经网络与深度学习的基础。

◎ 朴素贝叶斯模型：基于贝叶斯理论的分类模型，根据特征性质又分为三类。

◎ 支持向量机：在高维空间中构建超平面的分类和回归方法。

◎ 高斯过程：在时间等连续域上工作的统计学模型，在给出分类/回归结果的同时提供标准差等不确定程度估计。

◎ 决策树：预测结果最易于人类理解的机器学习模型。

◎ 集成学习：采用多个分类器对数据集进行预测，综合它们的结果从而提高整体泛化能力。

◎ 多标签与多输出：使用 Scikit 提供的工具在一个模型上预测多个标签或输出。

注意：本章实践代码依赖 scikit-learn 工具包，实践前可先用 pip3 命令进行安装。

3.1 线性回归

在学习本章前建议先回顾第 1 章中关于有监督学习的讨论。本节从基本的最小二乘法线性预测开始，分析其不足，逐步引入 Ridge 回归与 Lasso 回归。

3.1.1 何谓线性模型

有监督学习是通过已知的样本产生预测模型的学习方法，任何有监督学习模型都可以被想象成一个函数： $\boldsymbol{y}=\boldsymbol{f}\left(\boldsymbol{x}_1,\boldsymbol{x}_2,\boldsymbol{x}_3,\ldots,\boldsymbol{x}_{\boldsymbol{n}}\right)$

其中 $\boldsymbol{x}_1,\boldsymbol{x}_2,\boldsymbol{x}_3\ldots x_n$ 是模型中的 n 维特征，$\boldsymbol{y}$ 是要预测的目标值/分类。当 $\boldsymbol{y}$ 是可枚举类型时该模型解决的就是分类（classification）问题，当 $\boldsymbol{y}$ 是连续值时该模型解决的则是回归（regression）问题。

初等数学中就有线性回归（Linear Regression）的概念，在机器学习中它被用来解决学习特征与目标都是连续值类型的问题。线性模型将有监督学习模型定义为类似公式（3-1）的多项式函数：

$$\boldsymbol{y}=w_0+w_1\boldsymbol{x}_1+w_2\boldsymbol{x}_2+w_3\boldsymbol{x}_3+\ldots+w_{\boldsymbol{n}}\boldsymbol{x}_{\boldsymbol{n}} \qquad (3\text{-}1)$$

线性模型学习的目标就是确定 $w_0,w_1,w_2,w_3\ldots w_n$ 的值，使 $\boldsymbol{y}$ 可以根据特征值直接通过该函数计算得到；沿用初等数学的术语，在这些模型参数中 w_0 称为截距（intercept），其他 $w_1,w_2,w_3\ldots w_n$ 被称为回归系数（coefficient）。

然而有一些看上去与公式（3-1）大相径庭的多项式，比如：

$$y=2w_0+w_1x^1+w_2x^2+w_3x^3+\ldots+w_nx^4$$

$$y=w_0+w_1\sin\left(x\right)+3w_2\cos\left(x\right)+5w_3\sin\left(x_2\right)+\ldots+w_n\cos\left(x_n\right)$$

…

如上这些最终都可以转换为公式（3-1）的形式，所以它们其实也都是线性模型。相对于线性模型的其他模型就是所谓的非线性模型，非线性模型有更多形式，比如：

$$y = w_0 + \sin(w_1)x_1 + \cos(w_2)x_2 + \exp^{w_3 x_3} + \ldots$$

$$y = w_0 + \log(w_1)x_1 + \cos(w_2)x_2 \sin(w_3)x_3 + \ldots$$

$$\ldots$$

这些函数都不是线性模型，它们之间如果进行组合嵌套会产生更复杂的非线性函数。可以得出这样的结论：是否是线性模型取决于其被求系数 $w_0 \ldots w_n$ 之间是否为线性关系，而与样本特征变量 $x_0 \ldots x_n$ 的形式无关。

对非线性模型的学习通常无法通过像线性模型这样进行固定公式的参数逼近，需要使用带核函数的 SVM、神经网络等更复杂的模型。

3.1.2　最小二乘法

最简单的线性模型算法就是最小二乘法（Ordinary Least Squares，OLS），它通过最小化样本真值与预测值之间的方差和来达到计算出 $w_1, w_2, w_3 \ldots$ 的目的，即：

$$\operatorname{argmin}\left(\sum(\hat{\boldsymbol{y}} - \boldsymbol{y})^2\right) \tag{3-2}$$

其中 $\hat{y}$ 是样本预测值，y 是样本中的真值（ground truth）。在一维特征情况下其效果如图 3-1 所示，其中数据点是样本数据，实线线段是计算 w_0、w_1 后产生的线性模型，两条虚线是 w_0、w_1 取其他值时产生的模型举例。可以将 OLS 结果想象成在无数条任意虚线中找到这条最能逼近样本数据的一条直线的算法。

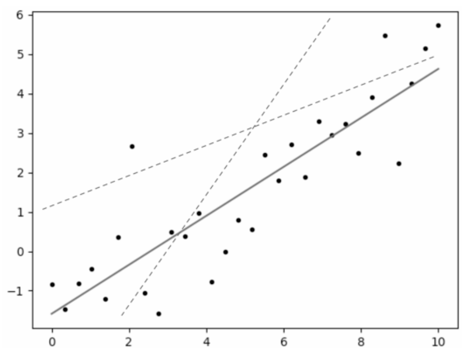


图 3-1　最小二乘法线性模型在一维特征情况下的效果

最小二乘法的具体计算原理涉及线性代数的较多推导，这里不展开叙述。开发者可以直接使用 sklearn.linear_model 中的 LinearRegression 对象进行训练与预测，比如：

```
>>>from sklearn import linear_model

>>>x = np.array([[0, 1], [3, -2], [2, 3]])                # 训练样本特征
>>>y = np.array([0.5, 0.3, 0.9])                          # 训练样本目标值

>>>reg = linear_model.LinearRegression()                  # 最小二乘法回归对象
>>>reg.fit(x, y)                                          # 训练、拟合

>>>print("intercept_: ", reg.intercept_)                  # 读取截距
intercept_:  0.366666666667
>>>print("coef_: ", reg.coef_)                            # 读取回归参数
coef_:  [ 0.06666667  0.13333333]

>>>reg.predict([[1, 2], [-3, 2]])                         # 预测
[ 0.7   0.43333333]
```

代码解析如下：

◎ 用 x、y 变量定义了三个样本数据，x 变量是一个 3×2 数组，因此每个样本包含两个特征。

◎ 初始化 LinearRegression 对象 reg 后，就可以调用其上的 fit()方法进行训练。

◎ 训练之后，对象 reg 就成为了一个线性模型，可以通过属性 intercept_和 coef_读取模型参数。

◎ 调用模型上的 predict()方法可以使用模型进行预测，该函数的参数是一个二维数组，意味着可以同时传入多组数据进行批量预测。本例中进行了两组预测。

通过读取 intercept_和 coef_属性，可以知道上例代码中拟合到的线性模型是：

$$y = 0.366666666667 + 0.06666667x_1 + 0.13333333x_2$$

另外，代码中使用的 fit()和 predict()函数是 scikit-learn 中所有模型的训练和预测方法，本书后续不再强调。

3.1.3 最小二乘法的不足

OLS 似乎已经可以解决所有线性模型回归的问题，但为什么还会出现 Ridge、Lasso

等模型呢？现在来看一下在数据特征维度逐渐增加的情况下 OLS 的表现如何：

```
def make_data(nDim):
    x0 = np.linspace(1, np.pi, 50)                          # 一个维度的特征
x = np.vstack([[x0,], [i**x0 for i in range(2, nDim+1)]])#nDim 个维度的特征

y = np.sin(x0) + np.random.normal(0,0.15,len(x0))          # 目标值
return x.transpose(), y

x, y = make_data(12)
```

以上代码定义的 make_data() 函数用于制造 50 个多维特征样本数据。其中 x0 是一个长为 50 的等差数列，x 是 50 个 nDim 维度的特征，x 中的每一个特征都与 x0 相关，因此这是一组特征之间严重相关的样本。

通过执行 make_data() 函数，赋值了 x、y 变量分别保存 50 个样本的 12 维特征与目标值。现在写一段代码，分别在特征维度为 1、3、6、12 的情况下进行 OLS 训练并查看模型参数：

```
def linear_regression():
    dims = [1,  3,  6, 12]                              # 要训练的维度

    for idx, i in enumerate(dims):
        plt.subplot(2, len(dims)/2, idx+1)
        reg = linear_model.LinearRegression()

        sub_x = x[:, 0: i]                              # 取 x 中前 i 个维度的特征
        reg.fit (sub_x, y)                              # 训练
        plt.plot(x[:,0], reg.predict(sub_x))            # 绘制模型
        plt.plot(x[:,0], y, ".")
        plt.title("dim=%s"%i)

        print("dim %d :"%i)
        print("intercept_: %s"% (reg.intercept_,))      # 查看截距参数
        print("coef_: %s"% (reg.coef_,))                # 查看回归参数
    plt.show()

linear_regression()
```

由于代码中 x0 的取值范围是[1, π]，目标值由对特征 x0 调用 sin()函数加随机噪声而来，因此期望中的模型应该是 sin 曲线的[1, π]段。真实的训练结果如图 3-2 所示。

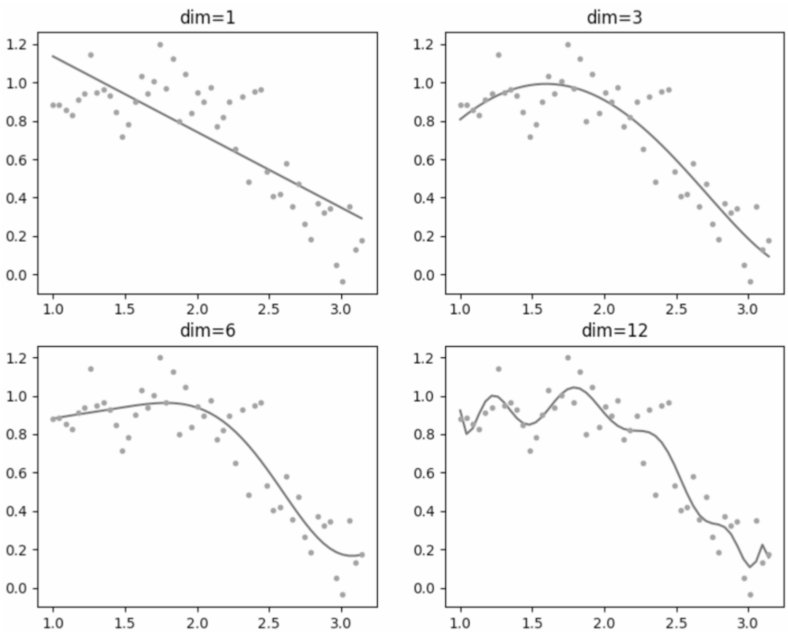


图 3-2　OLS 多维拟合比较训练结果

从图中观察：

◎　dim=1 时，模型是直线，存在拟合不足。

◎　dim=3、dim=6 时，比较接近 sin()曲线的[1, π]段。

◎　dim=12 时，曲线过于弯曲，存在过度拟合。

至此，凭直觉已经可以隐约感觉到，在 OLS 中随着特征维度的增加会出现线性模型的过度拟合。现在通过查看用不同维度执行四次训练后的模型参数来解释产生该现象的原因：

```
dim 1 :
intercept_: 1.6000369042
coef_: [-0.40280706]

dim 3 :
intercept_: 1.09916979162
coef_: [ 1.24750259  -0.82329499  0.07210458]

dim 6 :
intercept_: -201.714625599
coef_:  [  -99.17988853     458.5751755     -476.27789913   338.93929074
```

```
-136.67883365
      23.53301593]

    dim 12 :
    intercept_: 17334169.6061
    coef_:   [      3.91466134e+06      -1.31482075e+08        6.90338359e+08
-2.28706088e+09
      4.61840005e+09  -5.32540888e+09   2.21237965e+09   2.72873551e+09
     -4.97510932e+09   3.51369464e+09  -1.24376509e+09   1.81942377e+08]
```

以上数据是每次训练后打印的模型参数，套入线性模型公式，这些模型分别是：

$y_{\text{dim=1}} = 1.6000369042 - 0.40280706\,x_1$

$y_{\text{dim=3}} = 1.09916979162 + 1.24750259\,x_1 - 0.82329499\,x_2 + 0.07210458\,x_3$

$y_{\text{dim=6}} = -201.714625599 + 458.5751755\,x_1 - 476.27789913\,x_2 + 338.93929074\,x_3 + \ldots$

$y_{\text{dim=12}} = 17334169.6061 + 3.91466134e{+}06 - 1.31482075e{+}08\,x_2 + 338.93929074\,x_3 + \ldots$

一个明显的趋势是，随着特征维度的增加，模型求得的参数 w_0、w_1、$w_2 \ldots w_n$ 的值也显著增加。有兴趣的读者可以尝试更高维度的 OLS 模型，一定也是符合这个规律的。

产生这个现象的原因是 OLS 始终试图最小化公式（3-2）的值，因此为了更好地拟合训练数据中很小的 x 值差异产生的较大 y 值差异，必须使用较大的 w 值。而越来越大的 w 值在测试数据上反映出的结果则是任何一个特征微小的变化都会导致最终预测目标值的大幅度变化，即过度拟合。

3.1.4　岭回归

岭回归（Ridge Regression）是俄罗斯科学家 Tikhonov 提出的对 OLS 的改进（因此也被称为 Tikhonov Regularization），它通过改变回归目标函数，达到了控制回归参数值随着维度疯狂增长的目的。新的回归目标函数是：

$$\text{argmin}\left(\sum(\hat{y}-y)^2 + \alpha\sum w^2\right)$$

其与普通 OLS 的差别在于将 $\alpha\sum w^2$ 加入了最小化目标，其中 α 是一个可以调节的超参数，w 是线性模型中的所有参数。

注意：公式中的 $\alpha\sum w^2$ 被称为 L2 惩罚项（L2 Penalty）。

在 scikit-learn 中的 linear_model.Ridge 类实现了岭回归算法，用其改造之前的代码：

```
def ridge_regression():
    alphas = [1e-15, 1e-12,1e-5 ,1,]                    #α参数

    for idx, i in enumerate(alphas):
        plt.subplot(2, len(alphas)/2, idx+1)
        reg = linear_model.Ridge(alpha=i)               # 岭回归模型

        sub_x = x[:, 0: 12]                             # 取全部 12 维特征
        reg.fit (sub_x, y)                              # 训练
        plt.plot(x[:,0], reg.predict(sub_x))
        plt.plot(x[:,0], y, ".")
        plt.title("dim=12, alpha=%e"%i)

        print("alpha %e :"%i)
        print("intercept_: %s"% (reg.intercept_,))
        print("coef_: %s"% (reg.coef_,))
    plt.show()

ridge_regression()
```

Ridge 与原来的 LinearRegression 类使用方法几乎一样，只是在初始化对象的时候需要传入超参数 α，代码执行效果如图 3-3 所示。

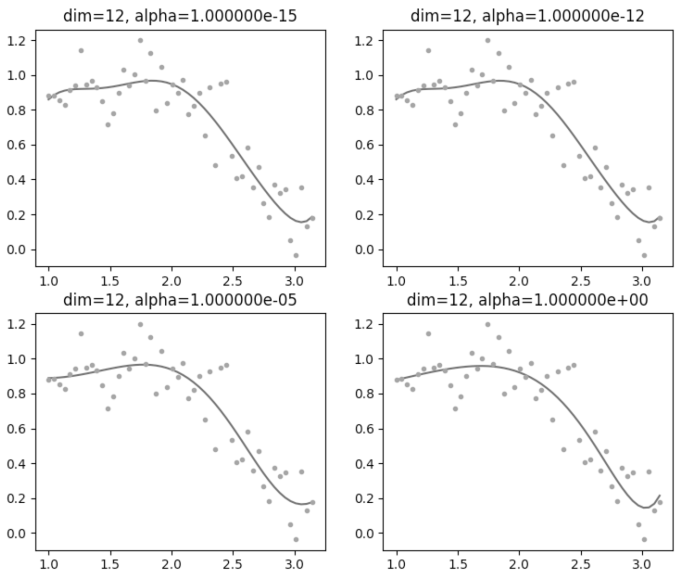


图 3-3　岭回归算法执行效果

本例的全部四次训练使用的都是在 OLS 中出现严重过度拟合的 12 维特征样本数据，可以观察到 Ridge 产生的线性模型确实对噪声的压制性更强。查看训练得到的参数：

```
    alpha 1.000000e-15 :
    intercept_: 2960.68429558
    coef_: [   994.49106665  -9447.70714758  15955.47832312 -15386.35363
       5100.32046732   2402.53302516   -270.627862    -2446.18545139
        735.04582132    808.43729599   -477.29287419     59.8046966 ]

    alpha 1.000000e-12 :
    intercept_: 2702.6817954
    coef_: [   907.57771514  -8598.82832513  14436.70378338 -13803.51159649
       4488.91024962   2163.86071992   -205.97077278  -2152.10703353
    592.8448356     736.87130945   -415.98716334     49.56123709]

    alpha 1.000000e-05 :
    intercept_: -25.5320040614
    coef_: [-21.33421403  19.83609031  23.60342591   1.07804025 -16.99640134
     -15.57547995   0.44679009  15.44290222  14.79523838  -3.99721987
     -21.68250754  10.83589165]

    alpha 1.000000e+00 :
    intercept_: 0.750498832343
    coef_: [  0.06979822   0.0564411     0.0659139     0.03927705   0.00071978
-0.02932411
     -0.03885076 -0.02551643  0.00330598  0.03089901  0.03172013 -0.02804581]
```

相比 OLS 模型的 12 维特征模型结果，模型参数 w 显著降低。并且 α 参数的大小与训练结果的回归参数呈反向关系：α 越大，回归参数越小，模型越平缓。

3.1.5　Lasso 回归

显然 Ridge 模型已经解决了多特征情况下回归模型参数太大、导致过度拟合的问题。但在 Ridge 模型中，无论将 α 设成多大，回归模型参数都只有非常小的绝对值，很难达到零值。这样造成的一个结果是：可能有很多特征对最终预测结果的影响微乎其微，但还是不得不将其加入模型计算中。在大数据系统中，这会对数据的产生、存储、传输、计算等

各环节产生较大浪费。

而 Lasso 则是一个可以将不重要的特征参数计算为零的模型，其目标函数形式是：

$$\text{argmin}\left(\sum(\hat{y}-y)^2+\alpha\sum|w|\right)$$

虽然其形式与 Ridge 差别不大，只是把惩罚项中的 w^2 替换为 $|w|$，称为 L1 Penalty，但其惩罚效果却比 L2 严厉很多：可以产生稀疏（spare）回归参数，即多数回归参数为零。

注意：L1 与 L2 的概念在很多其他模型中都有类似于 Lasso 与 Ridge 在线性模型上的应用，它们也被称为规则化（regularization）手段，其目的都是为了防止被求参数 w 取值过大。

在 scikit-learn 中的 linear_model.Lasso 类实现了 Lasso 回归，用其改造之前的代码：

```
def lasso_regression():
    alphas = [1e-10, 1e-3, 1 ,10,]                    # α 列表
    for idx, i in enumerate(alphas):
        plt.subplot(2, len(alphas)/2, idx+1)
        reg = linear_model.Lasso(alpha=i)             # 初始化 Lasso 对象
        sub_x = x[:, 0: 12]                           # 取全部 12 维特征
        reg.fit (sub_x, y)                            # 训练
        plt.plot(x[:,0], reg.predict(sub_x))
        plt.plot(x[:,0], y, ".")
        plt.title("dim=12, alpha=%e"%i)

        print("alpha %e :"%i)
        print("intercept_: %s"% (reg.intercept_,))
        print("coef_: %s"% (reg.coef_,))
plt.show()

lasso_regression()
```

以上代码与 ridge_regression 几乎一样，只是使用了不同的回归对象和 α 参数。在 12 维特征数据上进行实验的效果如图 3-4 所示。

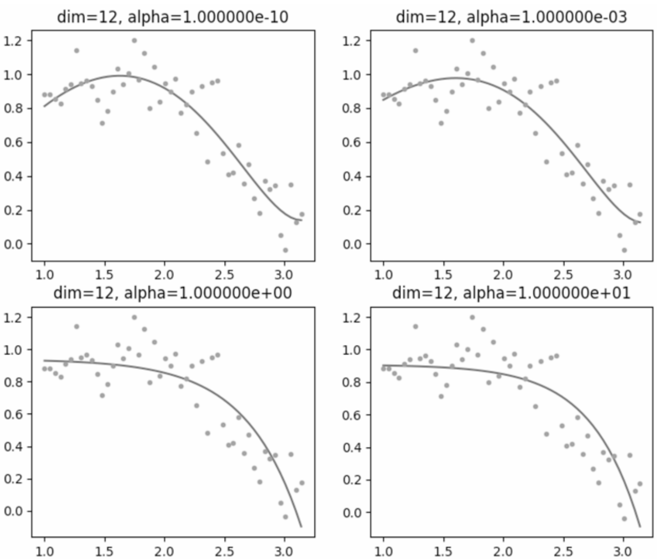


图 3-4　Lasso 回归实验效果

依旧观察其参数：

```
    alpha 1.000000e-10 :
    intercept_: 0.761691946916
    coef_:   [      8.20269619e-01      -3.59464864e-01      -1.22347228e-02
-4.07236510e-05
       3.79198458e-04   1.99678658e-04   8.31171300e-05   2.71782805e-05
       2.30143600e-06  -8.01178916e-06  -1.16466523e-05  -1.22803701e-05]

    alpha 1.000000e-03 :
    intercept_: 0.673421994246
    coef_:   [      3.36163706e-01        0.00000000e+00      -3.36009245e-02
-9.78056041e-03
      -1.48435852e-03  -1.94049971e-04  -0.00000000e+00   4.44683023e-05
       5.91816723e-05   4.24009139e-05   2.93955079e-05   2.03180749e-05]

    alpha 1.000000e-00 :
    intercept_: 0.93151868619
    coef_:   [      0.00000000e+00        0.00000000e+00      -0.00000000e+00
-0.00000000e+00
      -0.00000000e+00  -0.00000000e+00  -0.00000000e+00  -1.00147439e-03
      -3.42753604e-04  -5.97574359e-05  -0.00000000e+00  -0.00000000e+00]
```

```
alpha 1.000000e+01 :
intercept_: 0.904799653777
coef_: [-0.        -0.        -0.        -0.        -0.        -0.        -0.
 -0.        -0.        -0.        -0.00026711 -0.00025518]
```

随着 α 参数的增大，有越来越多的回归参数被置为 0。由此，Lasso 回归达到了压缩相关特征的目的。

思考：OLS、Lasso、Ridge 的优缺点对比。

3.2 梯度下降

梯度下降（Gradient Descent）是很多机器学习算法中的一个基本概念，用于迭代寻找函数最优解。在本节介绍梯度下降和它的分支——随机梯度下降。

与本章中的其他主题不同，梯度下降不是某个固定的算法模型，它是一种在大数据领域中用于求解问题的常用思想，是神经网络、深度学习领域的基础概念之一。本节介绍梯度下降的基本思想和 scikit-learn 中用梯度下降解决线性问题的开发实践。

3.2.1 假设函数与损失函数

梯度下降是一种因为计算机存储及运算能力限制而不得不启用的逐步逼近、迭代求解的方法，在理论上它不保证求得最优解，算法原理如图 3-5 所示。

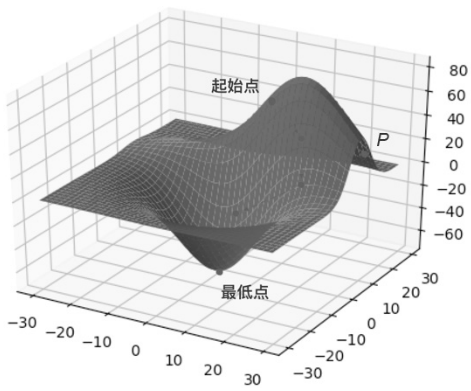


图 3-5　梯度下降算法原理

假设有三维曲面表达的函数空间，其长(x)、宽(y)轴为自变量，高轴(z)是因变量，用梯度下降法求解因变量最低点的步骤如下。

◎ 任取一点作为起始点。

◎ 查看在当前点向哪个方向移动能得到最小的 z 值，并向该方向移动。

◎ 重复该步骤，直到无法找到更小的 z 值，此时认为达到最低点。

受起始点和目标函数特性的影响，有时梯度下降法无法找到全局最优点（比如从图中 P 点出发很难找到最低点），但使用该方法求解能得到比 OLS 更快的求解速度，使得其在大样本学习中被广泛应用。

注意：此处说的“大样本”与统计学中的相同术语含义略有不同。这里指的是训练样本数量特别多、且特征维度特别多的机器学习场景。

根据上述求解原理，在动手开发之前需要熟知如下几个梯度下降求解算法概念。

◎ 步长（learning rate）：是在每一步梯度下降时向目标方向前行的长度，也就是图 3-5 中各步骤点之间的距离。不同的场景中步长的选择需要实验和权衡，步长越长，在陡峭区域下降越快，但在平缓区域容易出现反复抖动而找不到最优点；步长越短越不易产生抖动，但容易陷入局部最优解。

◎ 假设函数（hypothesis function），由特征产生目标变量的函数，常用 $h()$表示。对于线性模型，假设函数就是函数 $y = w_0 + w_1x_1 + w_2x_2 + \ldots + w_nx_n$ 。神经网络等其他模型有复杂得多的非线形假设函数，但其中一定包含梯度下降需要寻找的参数组合 w_0、w_1、$w_2 \ldots w_n$ 。

◎ 损失函数（loss function），常用 $J()$表示：可以想象成是给任意参数组合 w_0、w_1、$w_2 \ldots w_n$ 打分的函数。通过损失函数，图 3-5 中的每个点才知道周围的哪些点比自己更接近目标值。一个常用的损失函数是 $\sum(\hat{y} - y)^2$，代表所有样本评估值与知识值的方差和，该值越小则该组 w_0、w_1、$w_2 \ldots w_n$ 越好。

此时如果回顾上一节，其实当时已经围绕着假设函数和损失函数讲解了线性模型，此处只是给出了它们在梯度下降语境中的正式名称。

说明：用损失函数判断向周围哪个方向移动的原理是计算损失函数的偏导数向量，由微积分知识可知：该向量就是损失函数增长最快的方向，而其反方向则是以最小化损失函数为目标时需要前进的方向。

3.2.2 随机梯度下降

虽然普通梯度下降已经是一种不保证求得非全局最优解的权衡方法，但是在大样本情景下它还是不够快。普通梯度下降的损失函数是 $\sum(\hat{y}-y)^2$，所以在每次求损失值时都需要遍历所有样本数据后才能完成一次迭代，迈出小小的一步。

而所谓的随机梯度下降（Stochastic Gradient Descent，SGD）在损失函数计算时不遍历所有样本，只采用单一或小批量样本的方差和作为损失值。这样产生的结果是，每次迭代计算的速度非常快，通过每次随机选用不同的样本进行迭代达到对整体数据的拟合。SGD 的效果如图 3-6 所示。

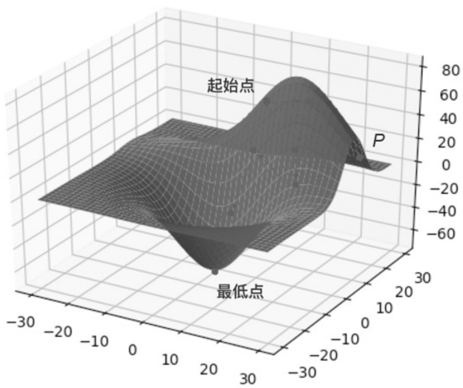


图 3-6　SGD 的效果

随机梯度下降与普通梯度下降的效果区别主要在于：

◎　迭代次数比原来有明显增加，但是由于每次只计算很少样本，总体时间非常短。

◎　由于数据样本中存在噪声，每次迭代的移动方向不一定是“正确的”；但由于迭代次数的增加，总体的移动期望仍然朝正确的方向运行。

◎　由第二点带来的好处是，本来在普通梯度下降中无法找到全局最优解的 *P* 点有更大的可能性通过“非正确移动”越过高点，从而找到全局最优解。

因此随机梯度下降在大多数情况下无论在运行速度还是结果正确性上都优于普通梯度下降。读者如果对 SGD 在结果有效性上的详细推导和证明有兴趣，可以参考 L’eon Bottou 的论文 *Large-Scale Machine Learning with Stochastic Gradient Descent*。

3.2.3　实战：SGDRegressor 和 SGDClassifier

scikit-learn 实现了随机梯度下降的线性模型分类器 SGDClassifier 和回归器 SGDRegressor，使用它们可以学习超大规模样本，即样本数量大于 10^5 且特征维度数高于 10^5。

1. SGDRegressor

举例学习 SGDRegressor 的调用方法：

```
>>>from sklearn.linear_model import SGDRegressor

>>>X = [[0, 0], [2, 1], [5, 4]]                                # 样本特征
>>>y = [0, 2, 2]                                               # 样本目标分类

>>>reg = SGDRegressor(penalty="l2", max_iter=10000)
>>>reg.fit(X, y)

>>>reg.predict([[4, 3]])
[1.97810841]

>>>reg.coef_                                                   # 查看回归参数
[1.56579508 -1.47844641]

>>>reg.intercept_                                              # 查看截距
[0.15026733]
```

这是一个用 SGDRegressor 进行回归计算的例子，其训练和预测方法与其他线性模型完全一样，在训练后仍然可以读取模型 coef_和 intercept_的属性。与其他线性模型不同的是，随机梯度下降对象有一些特有的初始化参数，比较重要的如下。

◎ penalty：损失函数惩罚项，取值“none”“l1”“l2”或“elasticnet”。请读者回顾 Ridge 和 Lasso 模型中的惩罚项概念，用它可以抑制模型过度拟合。“l2”和“l1”分别对应 Ridge 和 Lasso 模型的惩罚项，“elasticnet”是两者的综合。

◎ loss：损失函数类型，可取值“squared_loss”“huber”“epsilon_insensitive”或“squared_epsilon_insensitive”，对训练效果及速度有所影响。

◎ tol：在迭代后，损失函数的变化小于 tol 值时认为已经找到最优解。

◎ max_iter：最大迭代次数。有时模型在迭代中陷入反复抖动，无法找到满足 tol 停止条件的点，则只能利用 max_iter 作为停止迭代条件。

◎ shuffle：完成一轮所有样本迭代后是否需要洗牌（重新随机调整样本顺序）以开始下一轮迭代。

◎ n_jobs：训练中可利用的 CPU 数量。

◎ learning_rate：步长类型。可取值“constant”“optimal”“invscaling”，前者是固定步长，后两者为动态步长。动态步长是一种在初始时设置较大步长、随着训练进程逐渐减小步长的策略。有利于在训练初期跳跃出局部解，同时在后期避免抖动。

◎ eta0：在步长类型为“constant”或“invscaling”时的初始步长。

◎ fit_intercept：模型是否有截距，取值为 True 或 False。

2. SGDClassifier

SGDClassifier 与 SGDRegressor 的使用方法几乎一样：

```
>>>clf = SGDClassifier(penalty="l2", max_iter=100)     # 初始化分类器
>>>clf.fit(X, y)                                       # 训练

>>>clf.predict([[4, 3]])                      # 预测
[2]                                           # 预测结果与 SGDRegressor 不同
```

SGDClassifier 的对象初始化参数与 SGDRegressor 类似，不再重述。它们的不同点主要在于 predict()函数的预测结果。使用相同的数据对两者进行训练，SGDClassifier 预测的值一定是训练数据目标值之一，而 SGDRegressor 的预测值是假设函数直接的计算结果。这也是分类与回归的区别所在，scikit-learn 中的其他算法模型也有关系如同 SGDRegressor 与 SGDClassifier 的对偶封装类，后续不再强调。

3.2.4　增量学习

每次训练只需使用部分样本使得随机梯度下降具有增量学习（Incremental Learning）的特性，该特性是指一种可以边读数据边训练的拟合方法。每次训练是否能在之前训练成果基础上继续进行，是衡量大样本模型和强化学习模型的一条重要标准。

在 scikit-Learn 中提供了 partial_fit()函数接口，所有支持增量学习的模型都实现了该函数。SGD 的增量学习调用方法举例：

```
>>>from random import randint
>>>reg2 = SGDRegressor(loss="squared_loss", penalty="none", tol=1e-15)
```

```
    >>>X=np.linspace(0, 1, 50)                              # 50 个 x 值
    >>>Y = X/2+0.3 + np.random.normal(0, 0.15, len(X)) # 用 y=x/2+0.3 加随机数生成样本
    >>>X=X.reshape(-1, 1)

    >>>for i in range(10000):
          idx = randint(0, len(Y)-1)                        # 随机选择一个样本索引
    reg2.partial_fit(X[idx:idx+10], Y[idx: idx+10])    # 用 partial_fit()训练

    >>>print(reg2.coef_)                                    # 查看回归参数
    [0.49362945]
    >>>print(reg2.intercept_)                               # 查看截距
    [0.29178783]
```

本例代码在变量 X、Y 中建立了 50 个一维特征样本。与之前将所有样本放入 fit()中进行一次性训练不同，本例在循环中调用 partial_fit()函数进行训练，每次调用 partial_fit()使用的是随机选择的最多 10 个样本。

通过一定次数的迭代后，查看模型参数，发现当前的模型是：$y = 0.29178783 + 0.49362945x$，与生成样本时的公式相近。

3.3 支持向量机

支持向量机（Support Vector Machine，SVM）最初被用来解决线性分类问题，自 20 世纪 90 年代中期被加入核方法后能有效解决非线性问题。其优点主要是能适应“小样本数量、高特征维度”的数据集，甚至是特征维度数高于训练样本数的情况。

3.3.1 最优超平面

SVM 通过学习数据空间中的一个超平面达到二值分类目的。在预测中，在超平面一侧的被认为是一个类型的数据，在另一侧的被认为是另一种类型数据。

所谓的超平面在一维空间中是一个点，在二维空间中是一条线，在三维空间中是一个平面，在更高维度空间中只能被称为超平面。如图 3-7 所示，在普通线性可分问题中，符合分类要求的超平面会有无穷多个。

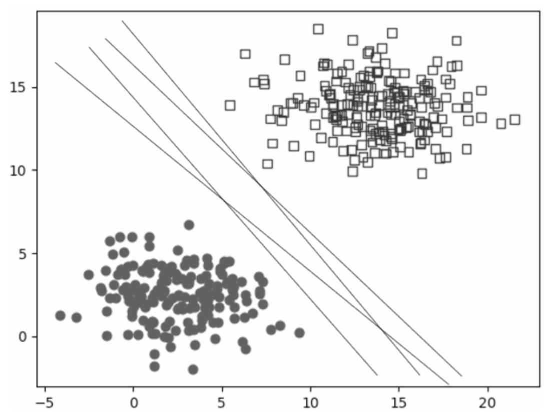


图 3-7　二值分类的无穷多个超平面

图 3-7 中的实心原点是一类数据，空心方块是一类数据。因为不同的数据分布在不同区域，人类凭直觉就能画出它们之间的无数个分类超平面。它们都能正确地将训练数据划分为两类，那么 SVM 会选取哪一个作为分类的标准（最优超平面）呢？

机器学习的目的从来不只是能正确匹配训练数据，而是要求分类器在新的测试数据上有良好表现。出于这个目标，在没有任何其他已知条件的情况下，分类器应该平均分配两类数据当中的空白区域，使得新数据能被分到它更靠近的那一侧。本例最优超平面如图 3-8（左）所示，如图 3-8（右）所示是加入一些新数据后的预测效果。

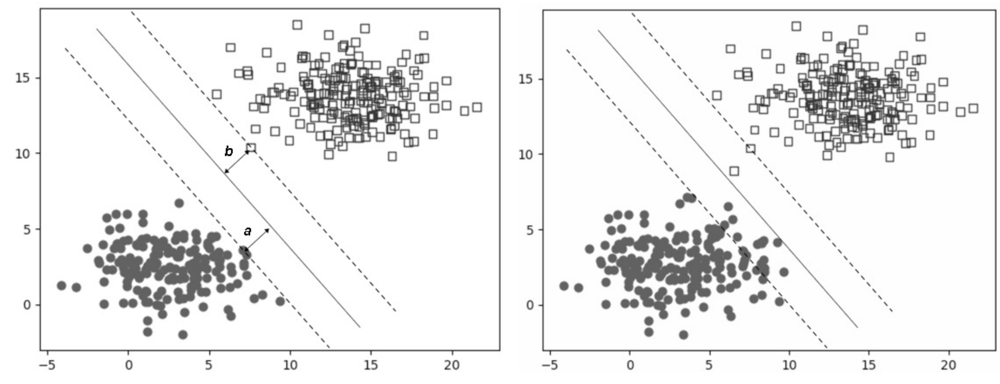


图 3-8　最优超平面（左）与预测效果（右）

图 3-8 中的虚线是两条最靠近空白区域点的平面，实线就是 SVM 选择的最优超平面（Optional Hyperplane），选择它们的依据是：

◎　无法找到其他绘制方法使得到两条虚线之间的距离更大。

◎　最优超平面到与两种类型距其最近的点有相等的距离，也就是在图 3-8（左）中的 $a=b$。

上述两条综合起来，使得“与超平面距离最小的数据点的距离”最大化。另外，图 3-8（左）中的线段 a 和 b 被称为超平面的间隔（Margin）；虚线上的训练数据点被称为支持向量（Support Vector），这也是 SVM 名称的由来。

3.3.2　软间隔

在很多时候训练集中会有噪声数据，或者问题本身带有不确定性，这时坚持使用最优超平面策略会产生过度拟合问题，如图 3-9 所示。

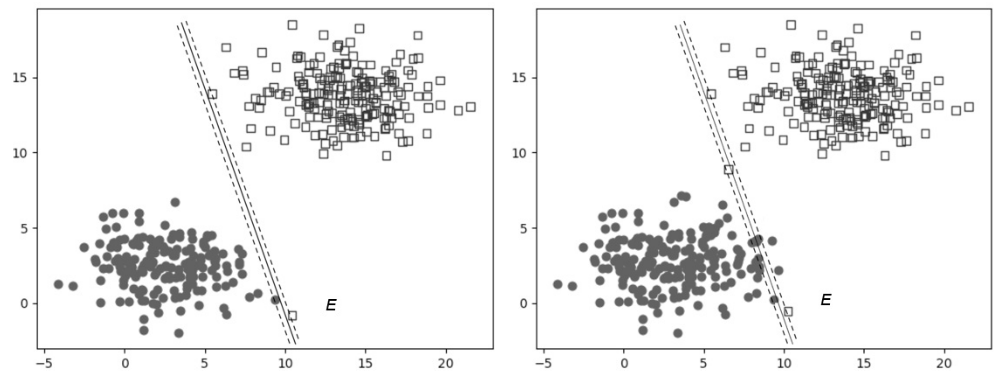


图 3-9　带噪声数据的训练（左）与预测（右）

在图 3-9 的中下部分有一个噪声数据点（空心方块点 E），导致 SVM 在计算最优超平面时只能取到一个很小的间隙，虽然其在训练数据集上（左图）仍然能正确分类两组数据，但是在右图加入测试数据后却表现不佳。

在 SVM 中所谓“软间隔（Soft Margin）”的概念解决了这类问题，它允许计算超平面时在训练集上存在错误数据，防止出现过度拟合。此时寻找超平面的问题变成了对如下两种条件的权衡：

◎　要尽可能正确地分类训练数据。

◎　margin 要尽可能大。

在类似图 3-9 的存在噪声的数据中，以上两个条件互相矛盾。SVM 模型中提供了松弛因子超参数“*C*”，使得开发者能够控制在训练时所倾向的条件。当配置的 *C* 参数较大时，SVM 倾向于超平面能严格分类训练数据；当 *C* 参数较小时，SVM 倾向于容忍更多训练数据分类错误而使 Margin 更大。

3.3.3 线性不可分问题

之前虽然讨论了噪声数据的处理，但训练数据大体上是能够在特征维度上进行线性分割的。如果训练数据的类型分布是如图 3-10 所示的情况，那么无论如何是找不到一个二维超平面较好地分割两类数据的，这就是所谓的非线性问题。

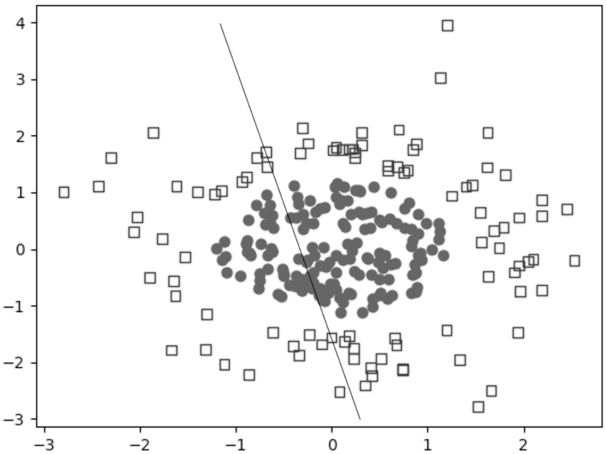


图 3-10 非线性分类问题

对于这样的数据集，可以在增加特征维度后找到分类超平面。假设将图 3-10 中的所有二维数据点通过如下公式映射到三维空间：

$$\begin{cases} x = x \\ y = y \\ z = \sqrt{(x^2 + y^2)} \end{cases}$$

其中 x 和 y 两个维度的数值不变，维度 z 的数值是 x 值与 y 值平方和的二次方根。映射后的数据如图 3-11（左）所示，可以发现图 3-11（右）中划分出的超平面①能正确分类数据。

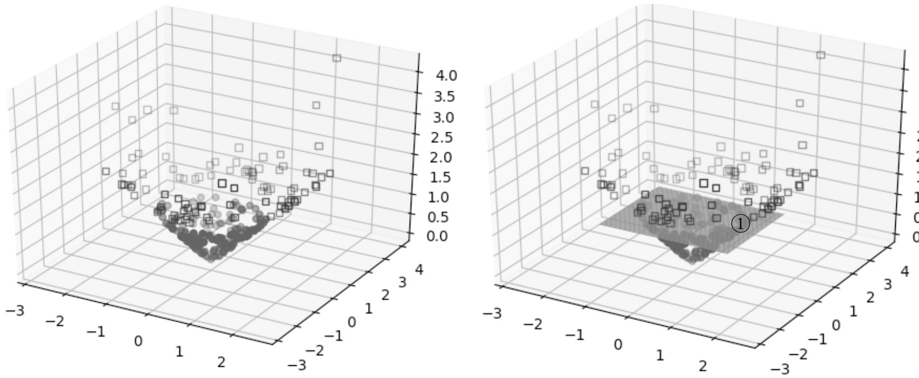


图 3-11　非线性问题映射到高维空间后的数据

因此可以想象：任何有限维度的非线性问题在更高维度的空间里总可以变化成线性可分问题。但是上述从二维到三维的映射公式是针对训练数据量身定制的，在 SVM 中是否需要对所有的数据都定制自己的映射函数呢？

答案当然是否定的，SVM 使用拉格朗日乘子法（Lagrange Multiplier）实现对超平面求解问题的升维。假设高维空间中最优超平面是形如 $y = w_0 + w_1x_1 + w_2x_2 + \ldots + w_nx_n$ 的线性方程，通过拉格朗日乘子法最后可以将求超平面参数 w 的目标转换为用高维中数据点向量两两点积（dot-product）值求解的二次规划问题。

说明：二次规划问题是指一种有约束情况下的求极值问题，SVM 具体如何使用该方法求解这里不再展开。

目前较好的解决二次规划问题的计算机算法是微软工程师 John Platt 于 1998 年提出的 SMO 算法，有兴趣的读者可阅读提出者本人的论文 *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*。

3.3.4　核函数

前文提到了 SVM 用拉格朗日乘子法将求超平面问题转换为用高维空间数据两两点积求解的二次规划问题。这意味着 SVM 无须真的将所有训练数据映射到高维空间，而只需知道这些数据在高维空间里两两之间的点积即可。

说明：点积是线性代数中的基本概念，设有两个向量 $\langle a_1, a_2, \ldots, a_n \rangle$、$\langle b_1, b_2, \ldots, b_n \rangle$，它们的点积就是标量 $a_1b_1 + a_2b_2 + \ldots + a_nb_n$。

核函数（Kernel Function）就是一种输入两个低维空间向量、返回高维空间点积的函数。使用 SVM 训练数据既可以选择一些通用的核函数，也可以自定义核函数。一些常用的核函数如下。

◎ 线性核（linear）：直接返回输入向量的点积，速度最快。因为实际上没有升维，适用于本身特征维度较高、样本数量很大的场景。

◎ 多项式核（polynomial）：$k(p,q)=(p\bullet q+1)\times d$，其中超参数 d 是提升到的维度。

◎ 高斯径向基核（Gaussian radial basis function）：$k(p,q)=\exp(-\gamma\|p-q\|^2)$，应用最广泛的 SVM 核，$\gamma$ 参数值越大越容易过拟合。

◎ Sigmoid 核：$k(p,q)=\tanh(a\times p\bullet q+r)$，其中 $\tanh(x)=\dfrac{e^x-e^{-x}}{e^x+e^{-x}}$，也是一种非线性核，有两个超参数 a、r 可以调整。

对于几个非线性核，很难给出每种核适用于什么场景的绝对条款，一方面多数情况下它们的表现差别不大，另一方面每种核在特定数据集上的表现仍依赖于调整合适的超参数。

3.3.5 实战：scikit-learn 中的 SVM

scikit-learn 没有直接实现 SVM 的复杂算法，而是通过调用两个成熟的 C++ SVM 计算库 libsvm、liblinear 提供 SVM 模型功能。在 sklearn.svm 中提供了三种分类/回归封装类。

◎ SVC/SVR：最普通的 SVM 分类器/回归器，可以通过 kernel 参数设置使用的核函数，使用 C 参数配置松弛因子。

◎ NuSVC/NuSVR：带有 nu 参数的分类器/回归器，nu 参数的作用与 C 参数类似，都是用来配置模型对训练数据拟合程度的。

◎ LinearSVC/ LinearSVR：使用 liblinear 库的线性核函数分类器回归器，其在模型中加入了线性回归惩罚参数。

用 SVC 类举例它们的使用方式：

```
>>>from sklearn import svm                          # 引入 SVM 包

>>>X = [[0, 0], [2, 2]]                             # 训练数据
```

```
>>>y = [1, 2]

>>>clf = svm.SVC(kernel="rbf)                      # 初始化使用径向基核的分类器
>>>clf.fit(X, y)                                   # 训练

>>>t = [[2, 1], [0, 1]]                            # 测试集
>>>clf.predict(t)                                  # 预测
[2 1]
>>>clf.decision_function(t)                        # 返回测试数据到超平面的距离
[ 0.52444566 -0.52444566]
```

其训练与预测方法和之前介绍的其他模型没有区别，值得关注的是代码最后调用的decision_function()函数，它可以返回输入的数据集与模型超平面之间的距离，用正负关系表示在超平面的哪一侧。毫无疑问，该距离的绝对值越大则分类的可靠性越高。

除了 kernel 参数意外事件，SVM 模型还有多个可配初始化超参数，表 3-1 总结了常用的 SVM 模型参数。

表 3-1　常用 SVM 模型参数

名称	解释	SVC/SVR	NuSVC/NuSVR	LinearSVC/LinearSVR
C	松弛因子，取值 0～∞	√		√
kernel	可取‘linear’‘poly’‘rbf’‘sigmoid’等	√	√	
degree	多项式核函数的维度参数	√	√	
gamma	‘poly’‘rbf’‘sigmoid’三种核的超参数	√	√	
tol	SMO 算法中的停止阈值	√	√	√
nu	取值 0~1，控制对训练数据拟合程度		√	
penalty	线性模型惩罚项，‘l1’或‘l2’			√

3.4　朴素贝叶斯分类

相对于前面的几种模型，朴素贝叶斯（Naive Bayes）是一种非常简单的分类算法。由于朴素贝叶斯基于概率模型，所以它的优点在于可以对预测标签给出理论上完美的可能性估计，但它也要求数据多维特征之间相互独立。

思考：为什么这个贝叶斯叫朴素贝叶斯？

3.4.1 基础概率

跳出机器学习的主题，概率本身其实是一个很有意思的话题。可能读者偶尔也会思考：双色球彩票的中奖率是多少？是否能够用地铁站里等待上车的人数预计下一辆车还有多久到达？

概率是对未来事件发生可能性的表述，它有如下关键概念。

◎ 概率值常用 P 表示，古典概率取值范围是 0～1 之间。比如：如果事件 A 一定不会发生，则有 $P(A) = 0$。

◎ 条件概率：用形如 $P(A|B)$的方式表达，其含义是"如果 B 已经发生，那么 A 发生的概率是多少"。

◎ 联合概率：是用来描述两个事件共同发生的概率，表达式 $P(AB)$、$P(A, B)$或 $P(A\cap B)$都是联合概率的表示符，其含义是"事件 A、B 同时发生的概率是多少"。

◎ 事件之间并的概率用 $P(A\cup B)$表示，其含义是"A 或 B 至少一个事件发生的概率"。

◎ 加法原理：$P(A\cup B) = P(A) + P(B) - P(A\cap B)$。

◎ 乘法原理：$P(A\cap B) = P(B)\bullet P(A|B) = P(A)\bullet P(B|A)$。

◎ 两事件独立的充分必要条件是 $P(A\cap B)= P(A)\bullet P(B)$，也就是事件 B 的发生对事件 A 是否发生没有任何影响，即 $P(A|B)=P(A)$。反之也是如此。

从概率定义本身出发，某个事件的概率可以通过公式 $P=\frac{\text{构成事件的元素数目}}{\text{整个空间的元素数目}}$ 计算获得，其也称为古典概率公式。但有些时候这样计算不够直观，需要通过其他事件的概率推导获得。举个例子，已知有两袋糖衣巧克力，它们分别装有不同颜色的巧克力。

◎ 袋 a：4 个红色，3 个绿色，3 个黄色巧克力。

◎ 袋 b：2 个红色，7 个绿色，11 个黄色巧克力。

假设巧克力的颜色在将其从袋子中取出后才能看到，请读者思考如下两个问题。

◎ 问题 1：从袋 a 中任意取出一个巧克力，它是红色的概率是多少？

◎ 问题 2：任取一袋，再从中取出一颗巧克力发现其为红色，那么它来自袋 a 的概率是多少？

相信读者可以很快想到问题 1 的答案，因为其可以通过古典概率直接计算获得，即

$P=\frac{\text{袋}a\text{中红色巧克力个数}}{\text{袋}a\text{中红色}+\text{绿色}+\text{黄色巧克力个数}}=\frac{4}{4+3+3}=\frac{2}{5}$。但对于问题 2 就不那么直观了，应该如何计算呢？贝叶斯定理正是为解决这类问题而来，其定义为：

$$P(A\mid B)=\frac{P(A)P(B\mid A)}{P(B)} \tag{3-3}$$

从公式可知其是描述了两个事件的条件概率之间的关系，在该公式中的每个元素又有其各自的名称：$P(A\mid B)$ 是后验概率、$P(A)$ 是先验概率、$P(B\mid A)$ 是似然度、$P(B)$ 是标准化常量。

现在回到取巧克力的情景，可以根据问题作定义：事件 A 是"取到袋 a"，事件 B 是"取出了一个红色巧克力"。所以问题二演变成了用贝叶斯公式求 $P(A|B)$，此时贝叶斯公式（3-3）中的各项元素就是：

◎　由于任意取了一袋，先验概率 $P(A)=\frac{1}{\text{巧克力袋数}}=\frac{1}{2}$。

◎　似然度 $P(B|A)$其实就是前面的问题 1，即 $P(B\mid A)=\frac{2}{5}$。

◎　$P(B)=P(\text{取袋}a)\times P(\text{从袋}a\text{中取到红色})+P(\text{取袋}b)\times P(\text{从袋}b\text{中取到红色})$
$=\frac{1}{2}\times\frac{4}{10}+\frac{1}{2}\times\frac{2}{20}=\frac{1}{4}$。

这样就可以很顺利地获得问题 2 的答案了，即：

$$P(A\mid B)=\frac{P(A)\times P(B\mid A)}{P(B)}=\frac{(1/2)\times(2/5)}{1/4}=\frac{4}{5}$$

所以虽然任意取到袋 a 的概率只有 0.5，但一旦发现从其中取出了一颗红色巧克力，那么它来自袋 a 的概率则会达到 0.8。

3.4.2　贝叶斯分类原理

朴素贝叶斯是应用贝叶斯定理进行有监督学习的一种分类模型。在该模型中，将贝叶斯定理公式 $P(A\mid B)=\frac{P(A)P(B\mid A)}{P(B)}$ 中的事件 A 看成被分类标签，事件 B 看成数据特征。由于通常数据特征是 n 维向量，所以 $P(B)$ 演变成了 n 个特征的联合概率。因此机器学习语境下的贝叶斯公式变成了：

$$P(y \mid x_1, x_2, \ldots, x_n) = \frac{P(y)P(x_1, x_2, \ldots, x_n \mid y)}{P(x_1, x_2, \ldots, x_n)}$$

其中 $x_1, x_2, \ldots, x_n$ 是数据的 n 维特征， y 是预测标签。

1. 预测

利用贝叶斯公式对某样本进行分类的伪代码如下：

```
for label in 所有标签：
    用贝叶斯公式计算在给定特征情况下出现该 label 的后验概率；

预测标签 ← 获得最高后验概率的 label
```

由于实际上计算了所有标签的后验概率，所以贝叶斯分类不仅可以提供该组特征最可能的标签，还能给出第 2、3、4……高可能性的标签，做出诸如“是苹果的可能性为 50%，是橘子的可能性为 20%，是桃子的可能性为 10%……”这样的预测形式。这是梯度下降、SVM 等分类器所做不到的，在某些应用场景中很有吸引力。

另外，由于对单条样本来说在计算所有标签的后验概率时公式中的 $P(x_1, x_2, ..., x_n)$ 保持不变，所以出于性能考虑在实践中无须将该标准化常量加入训练与预测的计算中。

2. 训练

对于训练来说关注的是贝叶斯公式中右侧的先验概率、似然度。

◎ 先验概率：可以由训练者根据经验直接给出，也可以自动计算：统计训练数据中每个标签的出现次数，除以训练总数就可以得到每个标签的先验概率 $P(y)$。

◎ 似然度：假定 n 维特征的条件概率符合某种联合分布，根据训练样本估计该分布的参数。比如对于高斯分布来说，学习的参数有期望值和方差。

3. 独立性假设

在如上所述训练与预测的原理中，难点最终归结为计算 n 维特征的联合分布。在此处朴素贝叶斯模型有一个约定，就是其假设所有 n 维特征之间是相互独立的。这大大简化了联合分布的计算难度，由此事件独立性的充分必要条件有：

$$P(x_1, x_2, \ldots, x_n) = P(x_1) \times P(x_2) \times \ldots \times P(x_n)$$

因此似然度函数为：

$$P(x_1, x_2, \ldots, x_n|y) = P(x_1 \mid y) \times P(x_2 \mid y) \times \ldots \times P(x_n \mid y)$$

这意味着可以将目标从计算联合分布的条件概率简化为计算各特征独自的条件概率，这也是模型名称中“朴素”（naive）的由来。

3.4.3　高斯朴素贝叶斯

训练似然度条件概率中的元素 $P(x_1 \mid y), P(x_2 \mid y) \ldots$ 的方法是假定特征符合某种分布，然后通过训练集数据估计该分布的参数。scikit-learn 在 sklearn.naive_bayes 包中根据假定的特征分布类型不同分别实现了三种模型，高斯朴素贝叶斯（Gaussian Naive Bayes）是其中之一。

高斯朴素贝叶斯使用的高斯分布就是常说的正态分布，假定所有特征条件分布符合：

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

其中 μ_y、σ_y^2 被学习的模型参数特征期望值与方差。具体学习方法涉及统计学的参数估计知识，此处不再展开。

在 scikit-learn 中实现高斯朴素贝叶斯的类是 GaussianNB，其调用方式举例如下：

```
>>>from sklearn import datasets                      # scikit-learn 资料数据库
>>>iris = datasets.load_iris()

>>>from sklearn.naive_bayes import GaussianNB  # 引入高斯朴素贝叶斯模型
>>>gnb = GaussianNB()                                # 初始化模型对象
>>>gnb.fit(iris.data, iris.target)                   # 训练

>>>gnb.class_prior_                                  # 查看模型先验概率
[ 0.33333333  0.33333333  0.33333333]                # 有三种标签，先各验 1/3

>>>gnb.class_count_                                  # 查看训练集标签数量
[ 50.  50.  50.]                                     # 训练集每种标签有 50 个样本

#由于数据有四维特征，并且有三种标签，所以训练后产生 3×4=12 个高斯模型
>>>gnb.theta_                                        # 查看高斯模型期望值 μy
[[ 5.006  3.418  1.464  0.244]
 [ 5.936  2.77   4.26   1.326]
```

```
 [ 6.588  2.974  5.552  2.026]]

>>>gnb.sigma_                                    # 查看高斯模型方差σy²
[[ 0.121764  0.142276  0.029504  0.011264]
 [ 0.261104  0.0965    0.2164    0.038324]
 [ 0.396256  0.101924  0.298496  0.073924]]
```

3.4.4 多项式朴素贝叶斯

多项式朴素贝叶斯（Multinomial Naive Bayes）是用多项分布（Multinomial Distribution）作为似然度概率模型的分类器。由于其实际衡量的是特征在不同标签之间的分布比例关系，所以特别适合文本分类场景（每个单词在不同类型文章中有一定的分布比例）。

先简单介绍多项分布的概念：假设某事件的结果有 k 种可能，在实验了 n 次之后每种结果出现了若干次，多项分布就是用于描述在实验了 n 次之后每种结果发生次数概率的分布。比如普通的骰子有 6 个面，那么掷骰子的结果就是符合 k=6 的多项分布。而掷了 n 次之后结果是 1 的次数有多少呢？这取决于骰子是否均匀，也就是骰子本身掷为 1 的概率是多少。所以在应用多项分布估算 n 次实验的结果之前，还要知道单次实验每种结果发生的可能性是多少，这就是多项分布的超参数。比如掷骰子的超参数<0.1、0.1、0.1、0.1、0.1、0.5>就具体化了一个多项分布，其代表的是一种极为不均匀的骰子，因为某面出现的概率达到了 50%。

设训练集中有 m 种分类标签，多项式朴素贝叶斯假定每个特征都符合参数是向量 $\langle\theta_{y_1},\theta_{y_2},\ldots,\theta_{y_m}\rangle$ 的多项式分布，向量中的每个值是：

$$\theta_{y_i} = \frac{N_{y_i} + \alpha}{N_y + \alpha \cdot \mathrm{M}}$$

其中 N_{y_i} 是特征 i 在当前标签中的总数，N_y 是当前标签所有特征的总数。超参数 α 是平滑先验，其目的是防止 N_{y_i} 为零从而导致 θ_{y_i} 也为零的情况发生。α 通常设为 1，也就是拉普拉斯平滑（Laplace Smoothing）。

为什么要用平滑参数防止 θ_{y_i} 为零呢？这是由训练样本的有限性造成的。对于出现概率很小的特征来说，没有出现在某标签的训练样本中并不代表其以后也永远不会出现，所以合理的方法是用平滑参数赋予其一个很小的概率值，而不是零。

scikit-learn 中实现多项式贝叶斯的类是 MultinomialNB，与 GaussianNB 不同的是其在模型初始化中多了参数 α，并且训练后模型中不再提供高斯特有的 theta_、sigma_参数。

3.4.5　伯努利朴素贝叶斯

伯努利朴素贝叶斯（Bernoulli Naive Bayes）使用伯努利分布（Bernoulli Distribution）作为似然度概率模型。所谓伯努利分布也称二值分布，用来描述一次实验只可能出现两种结果的事件概率分布。由于伯努利分布只能描述二值结果，因而在该学习模型中要求数据中的所有特征都是布尔/二值类型。

scikit-learn 中实现伯努利朴素贝叶斯模型的类是 BernoulliNB，它用如下公式计算第 i 个特征的似然度：

$$P(x_i \mid y) = P(i \mid y)x_i + (1 - P(i \mid y))(1 - x_i)$$

其中 $P(i \mid y)$ 是第 i 个特征在所有该标签训练数据中出现的比。

在调用方式上，BernoulliNB 与 MultinomialNB 稍有不同的地方是它有一个 binarize 参数。参数 binarize 被用来当作一个阈值，将非二值特征转化为二值，比如：

```
>>>from sklearn.naive_bayes import BernoulliNB

>>>clf = BernoulliNB(binarize=1)                          # 设置特征阈值为 1

#可以输入非二值特征，BernoulliNB 内部会用阈值 1 将其转换为二值
>>>X = [[0.3, 0.2], [1.3, 1.2], [1.1, 1.2]]
>>>Y = [0, 1, 1]
>>>clf.fit(X, Y)                                          # 训练
>>>clf.predict([[0.99, 0.99]])                            # 预测
[0]
```

因为代码中 binarize 的阈值为 1，所以特征[0.99, 0.99]被认为与[0.3, 0.2]是同一类标签，而与在数值上距离更近的[1.1, 1.2]为不同标签。

思考：回忆一下，自己记住了几个贝叶斯，它们有什么区别？

3.5　高斯过程

在机器学习领域，高斯过程（Gaussian Process）是一种假设训练数据来自无限空间、并且各特征都符合高斯分布的有监督建模方式。高斯过程是一种概率模型，无论是回归或分类预测都以高斯分布标准差的方式给出预测置信区间估计。

3.5.1 随机过程

高斯过程应用于机器学习已经有数十年的历史，它来源于数学中的随机过程（Stochastic Process）理论。随机过程是研究一组无限个随机变量内在规律的学科，本节举例说明“无限个随机变量”与机器学习建模的关系。

假设需要训练一个预测某城市在任意时间居民用电量的模型。简化期间，在该模型中可以用当前温度、年内天数（即一年中的第几天，取值从 1～365+1）、当天时间作为数据特征，将居民总用电量作为目标标签数值。用有监督学习的思维，首先需要收集历史用电数据，比如在 2017 年的每天中午 12:00 收集并记录数据，这样得到了一组包含 N=365 条数据的训练数据。

但显然居民用电在一天内的不同时间段是有变化的，如果仅仅中午的采样数据不能精确建模，那么可以改为每天采样两次。如果仍不能满足需求，可以增加到每小时采样、每分钟采样……可以发现，随着精度要求的增加采样的训练数据是可以无限增加的，采样数据如图 3-12 所示。

样本编号	特征向量 X			目标值 Y
	温度	年内天数	时间	用电量（万瓦特）
1	-1	1	01:50:00	318
...	...	...	...	...
499	23	180	12:00:00	754
...	...	...	...	...

图 3-12　用电量采样数据

如果把每次采样的目标值用电量 y 都看成一个随机变量，那么单条采样就是一个随机分布事件的结果，N 条数据就是多个随机分布采样的结果，而整个被学习空间就是由无数个随机变量构成的随机过程了！

一个不太容易想明白的问题是，为什么要把实实在在采样得到的 Y 值看成随机变量呢？这涉及两个方面：一是所有数据的产生本身就是随机的，试想自己每天开空调或灯光是否都有随性成分？另一方面，数据的采集是有噪声存在的，这部分在通信领域也叫白噪声，在统计学中叫系统误差。

对于这样的场景，无论如何不可能给出一个精确值的预测，即使给出后碰巧符合也只能说明运气不错。更合理的预测方式应该是一个置信区间预测，比如 705±30 的概率达到 95%。

3.5.2　无限维高斯分布

如果把每个随机变量都用高斯分布进行建模，那么整个随机过程就是一个高斯过程了。高斯过程能成为随机过程最广泛的应用之一是因为高斯分布本身的诸多优良特性。

1. 高斯分布的特点

机器学习开发者应该熟知以下内容。

◎ 可标准化：一个高斯分布可由均值 μ 和标准差 σ 唯一确定，用符号 $\sim N(\mu,\sigma)$ 表示。并且任意高斯分布可以转化为用 $\mu=0$ 和 $\sigma=1$ 的标准正态分布表达。

◎ 方便统计：高斯分布中约 69.27%的样本落在 $(\mu-\sigma,\mu+\sigma)$ 之间，约 95%的样本落在 $(\mu-2\sigma,\mu+2\sigma)$ 之间，约 99%的样本落在 $(\mu-3\sigma,\mu+3\sigma)$ 之间。

◎ 多元高斯分布（Multivariate Gaussian）：n 元高斯分布描述 n 个随机变量的联合概率分布，由均值向量 $\langle\mu_1,\mu_2,\ldots,\mu_n\rangle$ 和协方差矩阵 $\boldsymbol{\Sigma}$ 唯一确定。其中 $\boldsymbol{\Sigma}$ 是一个 $n\times n$ 的矩阵，每个矩阵元素描述 n 个随机变量两两之间的协方差。

◎ 和与差：设有任意两个独立的高斯分布 U 和 V，那么它们的和 $U+V$ 一定是高斯分布，它们的差 $U-V$ 也一定是高斯分布。

◎ 部分与整体：多元高斯分布的条件分布仍然是多元高斯分布，也可理解为多元高斯分布的子集也是多元高斯分布。

以上特性综合在一起使得高斯分布特别利于计算。下面马上利用这些特性给出高斯过程学习的训练与预测原理。

2. 再遇核函数

已经知道可以将高斯过程看成无限维的多元高斯分布，那么机器学习的训练过程目标就是学习该无限维高斯分布的子集——也是一个多元高斯分布的参数：均值向量 $\langle\mu_1,\mu_2,\ldots,\mu_n\rangle$ 和协方差矩阵 $\boldsymbol{\Sigma}$ 。

协方差矩阵 $\boldsymbol{\Sigma}$ 中的元素用于表征两两样本之间的协方差，这个“描述两两样本之间关系”的概念似曾相识。对，就是 SVM 中用于计算高维空间两两样本向量内积的核函数。此处核方法也应用在了协方差矩阵上，使得多元高斯分布也具有了表征高维空间样本之间关系的能力，也就是具备了表征非线性数据的能力。此时的协方差矩阵可以表示为：

$$\boldsymbol{\Sigma} = \boldsymbol{K}_{XX} = \begin{bmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & \cdots & k(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ \vdots & \ddots & \vdots \\ k(\boldsymbol{x}_N, \boldsymbol{x}_1) & \cdots & k(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{bmatrix}$$

其中符号 $\boldsymbol{K}_{XX}$ 表示样本数据特征集 X 的核函数矩阵，用 $k()$表示所选取的核函数，$\boldsymbol{x}_1, \boldsymbol{x}_2 \ldots \boldsymbol{x}_n$ 等是单个样本特征向量。和 SVM 一样，此处的核函数也需要开发者指定其形式：常用的仍然有径向基核、多项式核、线性核等。在训练过程中可以定义算法自动寻找核的最佳超参数。

3. 预测

学到这些参数后如何进行预测呢？设样本目标值是 Y，被预测的变量是 Y_*，由高斯分布的特性可知，由训练数据与被预测数据组成的随机变量集合仍然符合多元高斯分布，即：

$$\begin{pmatrix} Y \\ Y_* \end{pmatrix} \sim N\left(\begin{pmatrix} u \\ u_* \end{pmatrix}, \begin{pmatrix} \boldsymbol{K}_{XX} & \boldsymbol{K}_{X_*X} \\ \boldsymbol{K}_{XX_*} & \boldsymbol{K}_{X_*X_*} \end{pmatrix}\right)$$

其中 u_* 是待求变量 Y_* 的均值，$\boldsymbol{K}_{X_*X}$ 是样本数据与预测数据特征的协方差矩阵，$\boldsymbol{K}_{X_*X_*}$ 是预测数据特征的协方差矩阵。

由完美的多元高斯特性可知 Y_*，也满足高斯分布 $N(u_*, \boldsymbol{\Sigma})$，并且可以直接用公式推导出该分布的超参数：

$$\begin{cases} u_* = \boldsymbol{K}_{X_*X}^{\mathrm{T}} \boldsymbol{K}^{-1} Y \\ \boldsymbol{\Sigma} = \boldsymbol{K}_{X_*X_*} - \boldsymbol{K}_{X_*X}^{\mathrm{T}} \boldsymbol{K}^{-1} \boldsymbol{K}_{XX_*} \end{cases}$$

其中涉及若干矩阵转置、求逆、乘法等运算。开发者也许不用理会其中这些细节，只需知道该公式通过下面这些已知数据直接计算出预测数据期望值与方差：

◎ 训练数据的特征向量集 X 和目标向量 Y。

◎ 预测数据的特征向量集 X_*。

◎ 开发者选择的核函数 $k()$ 及在训练过程中优化的核超参数。

与大多数机器学习模型不同的是，由于高斯过程在预测过程中仍然需要用到原始训练数据，因此导致该方法通常在高维特征和超多训练样本的场景下显得运算效率低。但也正是因此，它才能提供它们所不具备的基于概率分布的预测。

4. 白噪声处理

在建模中已经知道随机过程需要考虑采样数据存在噪声的情况。用高斯分布的观点来看，就是在计算训练数据协方差矩阵 $\boldsymbol{K}_{XX}$ 的对角元素上增加噪声分量。因此协方差矩阵变为如下形式：

$$\boldsymbol{\Sigma} = \boldsymbol{K}_{XX} = \begin{bmatrix} k(x_1,x_1) & \cdots & k(x_1,x_N) \\ \vdots & & \vdots \\ k(x_N,x_1) & \cdots & k(x_N,x_N) \end{bmatrix} + \alpha \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 1 \end{pmatrix}$$

其中 α 是模型训练者需要定义的噪声估计参数，该值越大模型抗噪声能力越强，但容易产生拟合不足。

3.5.3　实战：gaussian_process 工具包

至此已经具备了足够的理论知识，可以使用 scikit-learn 的 gaussian_process 包做高斯模型数据开发了。

1. kernels

核函数的选择在高斯过程模型中扮演着重要角色，在 sklearn.gaussian_process.kernels 中以类的方式提供了若干个核函数，常用的如下。

- ◎ ConstantKernel：常数核，对所有特征向量返回相同的值，实际上是使模型忽略特征数据信息。
- ◎ DotProduct：点积核，返回特征向量点积，也就是线性核。
- ◎ RBF：径向基核，把特征向量提升到无限维以解决非线性问题。

此外使用如下类还允许不同核之间进行组合：

- ◎ Sum（$k1$，$k2$）：用两个核分别计算后将结果相加。
- ◎ Product（$k1$，$k2$）：两个核分别运算后结果相乘。
- ◎ Exponentiation（k，exponent）：返回核函数结果的指数运算结果，即 k^{exponent} 。

2. GaussianProcessRegressor 与 GaussianProcessClassifier

GaussianProcessRegressor 是高斯过程回归模型，GaussianProcessClassifier 是高斯过程分类模型，它们在使用方式上与其他模型略有不同的是，它们的预测函数 predict()可以有两个返回值，第一个是预测期望值，第二个是预测标准差。此外，它们还有几个高斯过程特有的初始化参数。

◎ kernel：核函数对象，即 sklearn.gaussian_process.kernels 中类的实例。

◎ alpha：为了考虑样本噪声在协方差矩阵对角量增加值，可以是一个数值（应用在所有对角线元素上），也可以是一个向量（分别应用在每个对角线元素上）。

◎ optimizer：可以是一个函数，用于在训练过程中优化核函数超参数。

◎ n_restarts_optimizer：optimizer 被调用的次数，默认是 1。

3. 实例

以下代码改编自 scikit-learn 官网的实例，是一个高斯回归模型训练与预测的完整演示。它首先用一个非线性函数 $y = x \cdot \sin(x) - x$ 生成带噪声的训练样本，然后用高斯过程模型模拟该函数，并比较模型预测结果与真实目标值。

```
import numpy as np
from matplotlib import pyplot as plt

from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF
from sklearn.gaussian_process.kernels import ConstantKernel as C, Product

def f(X):                                      # 原函数
    return X*np.sin(X)-X

X = np.linspace(0, 10, 20)                     # 20 个训练样本数据的特征值
y = f(X) + np.random.normal(0, 0.5, X.shape[0])# 样本目标值，并加入噪声
x = np.linspace(0, 10, 200)                    # 测试样本特征值

# 定义两个核函数，并取它们的积
kernel = Product(C(0.1) , RBF(10, (1e-2, 1e2)))

# 初始化模型：传入核函数对象、优化次数、噪声超参数
```

```
    gp = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=3,
            alpha=0.3)

    gp.fit(X.reshape(-1, 1), y)                                 # 训练
    y_pred, sigma = gp.predict(x.reshape(-1, 1), return_std=True) # 预测

    fig = plt.figure()                                          # 用 matplotlib 绘制结果
    plt.plot(x, f(x), 'r:', label=u'$f(x) = x\,\sin(x)-x$')
    plt.plot(X, y, 'r.', markersize=10, label=u'Observations')
    plt.plot(x, y_pred, 'b-', label=u'Prediction')
    plt.fill(np.concatenate([x, x[::-1]]),
            np.concatenate([y_pred - 2 * sigma,                 # 填充 μ±2σ 的置信区间
                           (y_pred + 2 * sigma)[::-1]]),
            alpha=.3, fc='b', label='95% confidence')

    plt.legend(loc='lower left')
    plt.show()
```

代码运行结果如图 3-13 所示。

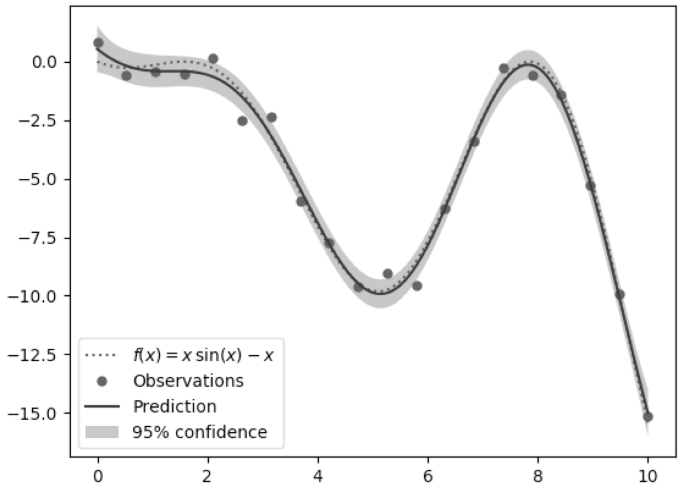


图 3-13　高斯过程回归预测结果

图 3-13 中的实心原点是训练数据，并且在整个区间均匀分布了 200 个测试样本：虚线是函数的真实值，实线是测试样本预测值，带条是预测值的$\pm2\sigma$范围。可以发现整体上测试样本的真实值虚线与预测值实线基本重合，只在左上角区域有较大偏差，但都在 95% 的置信区间内。

3.6 决策树

决策树（Decision Tree）是一个非常成熟的算法，目前最常用的三种决策树算法 ID3、C4.5、CART 均出现于 20 世纪 80 年代。虽然其理论基础比较简单，但由于训练后可以产生非常直观易懂的树形图，决策树至今仍被广泛应用。

3.6.1 最易于理解的模型

决策树最初被用来解决分类问题，目标是从大量的样本数据特征中找到分类决策路径。比如有一个银行 VIP 客户识别系统，定义了如表 3-2 所示的已有样本。

表 3-2 客户分类样本数据

特征向量 ***X***			目标值 ***Y***
年龄	月收入	存款	客户级别
20	30000	400	VIP
37	13000	0	普通
50	26000	0	普通
28	10000	3000	普通
31	19000	1500000	VIP
46	7000	6000	普通

虽然只是一组三维特征数据，普通人可能还是无法一下子识别出客户级别划分的依据是什么，但是通过训练可以获得如图 3-14 所示的决策树。

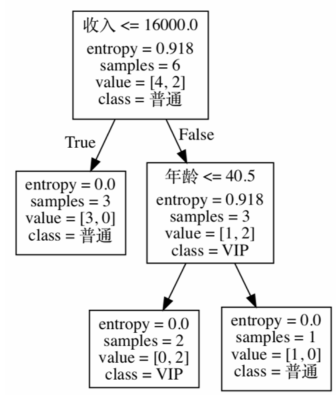


图 3-14 客户分类决策树

这是一棵 CART 二叉决策树，预测时从根开始每个非叶子字节执行一次某个特征的逻辑判断，当条件为真时走向左子树，否则走向右子树。如此走到叶子结点时就可以知道被预测的标签（图中叶子结点的 class 属性）。从本例训练的结果来看，实际只有两个特征在分类预测中起了作用：收入是否小于 16000，年龄是否小于 40.5。

在图 3-14 中每个结点还有另外三个属性。samples：本子树一共包含了多少训练数据；value：本子树训练数据中各种标签类型的样本数量，比如[4, 2]说明第一个类型的样本有 4 条，第二个类型的样本有 2 条；entropy：该结点的信息熵值。熵和信息增益是决策树训练过程中的核心概念。

3.6.2　熵的作用

在热力学中熵（entropy）被用来衡量系统的不稳定程度。信息熵的概念由信息论奠基人香农于 1948 年在论文《通信的数学原理》中提出，其目的是用于量化数字信息的价值。可能是由于人们生活中习惯于只对信息做定性判断，直到今天熵仍然不是一个被大众所熟知的概念。人们可能会说“这份情报很有价值”“总是收到垃圾短信”，但却很少思考该条信息好到何种程度或无用到什么地步。

1. 信息熵的定义

香农提出了量化信息的方式，即

$$\text{随机事件的熵} = H(P_1, P_2, \ldots, P_n) = -\sum_{i=1}^{n} P_i \cdot \log_2(P_i)$$

公式中 $P_1, P_2 \ldots P_n$ 是随机事件每种可能结果的发生概率，所以必有 $P_1 + P_2 + \ldots P_n = 1$。熵 $H()$的结果是一个大于等于零的值，熵越高说明事件的不确定性越大。而当有信息表明不确定性越大事件的结果时，该条信息的价值越高。

2. 熵的通俗解释

打个比方，假设有人从未来穿越回来告诉您两件事：a)下一次抛硬币的结果，b)下一次掷骰子的结果。哪条信息的价值更高呢？通过熵的公式可以有如下计算结果：

◎　假设硬币是均匀的，每面发生的概率是 1/2，则熵

$$H = -\left(\frac{1}{2} \times \log_2\left(\frac{1}{2}\right) + \frac{1}{2} \times \log_2\left(\frac{1}{2}\right)\right) = 1;$$

◎ 假设骰子是均匀的，每面的概率是 1/6，则熵 $H=-\left(\frac{1}{6}\times\log_2\left(\frac{1}{6}\right)+\ldots\right)\approx 2.59$。

因此抛硬币事件结果的信息价值是不如掷骰子结果的价值的。

现在想象另一种情况，如果老千制作了一枚不均匀的骰子，其中某一面发生的概率达到了 0.9，而另五面的概率都只有 0.02，那么获知此枚骰子掷出结果的信息熵的值是 $H=-\left(\frac{9}{10}\times\log_2\left(\frac{9}{10}\right)+\frac{1}{20}\times\log_2\left(\frac{1}{20}\right)\times 5\right)\approx 1.22$，所以该条信息的价值已经远小于均匀骰子了。这很好理解，因为即使没人告知，也可以断定该枚骰子的结果肯定是发生在概率达到 0.9 的那一面。通过上述例子，可以很通俗地解释信息熵了：

◎ 某随机事件结果的种类越多，则该事件的熵越大。

◎ 某随机事件的各种可能发生的结果概率越均匀，则该事件的熵越大。

3. 基尼指数（Gini index）

根据香农的论文，用信息熵衡量信息的价值有完美的理论依据。但有一个问题是：在当前计算机 CPU 架构中计算 $\log_2()$ 函数非常耗时，而在熵的计算公式中大量依赖该函数。因此出现了另一个衡量信息价值的指标——基尼指数：

$$\text{随机事件的基尼指数} = G(P_1,P_2,\ldots,P_n) = 1-\sum_{i=1}^{n}P_i^2$$

该公式有闭合的值域范围 0～1，数值越大表示事件越无序。基尼指数衡量信息价值的能力略逊于熵，但是它的公式由普通乘法和加法构成，并且有封闭的取值范围，因此非常适合在实际开发中使用。

4. 建树策略

决策树的训练是一个利用已有样本从根开始逐步挑选特征并建树的过程。比如最开始的任务是寻找用哪一个特征作为根结点，而选择的依据是在用该特征进行数据划分后得到的信息增益最大。

说明：在用某个特征将数据集划分到不同的子树后，所有子树信息价值（熵/基尼指数）的和必定小于等于原来整体数据集的信息价值，信息增益用来衡量减少的程度。

设有建树函数 build()，很容易写出递归训练决策树的算法思想。

```
def build(D=数据集，):
if D 中所有数据目标值 y 都相同：
    return                                         # 本数据集可以作为叶子结点
for i in D 中的所有特征：
    计算用 i 划分子树后获得的信息增益
if 所有的特征都没有大于零的信息增益：              # 已经无法再分
    return
被选择的特征 x = 具有最大信息增益的特征
for sub in 按照 x 划分子树后的数据集：
    build(sub)                                     # 递归寻找下一个决策特征
```

有了上述算法思想后还有很多细节要处理，比如：

◎　信息增益如果直接用划分前后的熵差计算，则会导致倾向于先分类取值比较多的特征（回想抛硬币与掷骰子哪个信息熵更高的例子）。

◎　连续值类型的特征如何计算？

◎　强制要求每个叶子结点只有一个目标值容易导致预测过度拟合，如何适当归并叶子结点——剪枝（prune）？

◎　叶子结点只能保存分类问题的目标值，那么如何用决策树处理回归问题？

根据对这些问题的处理策略不同，决策树家族又陆续衍生出了很多具体算法，目前有较大影响的是 ID3、C4.5 和 CART 算法。其中 ID3 只能使用熵的信息增益处理离散特征的分类问题；C4.5 在其中加入了：

◎　使用信息增益比的概念去除先选择多值特征的倾向。

◎　支持连续特征，在计算信息增益比之前首先将其离散化。

◎　在训练后检测训练集的正确分类比，并剪枝产生错误较多的叶子结点。

而 CART 算法主要的不同在于使用基尼系数代替熵进行信息增益计算、只使用二叉树、并提供了解决回归问题的能力，因此综合看来 CART 比另两种算法能适应更多的场景。

3.6.3　实战：DecisionTreeClassifier 与 DecisionTreeRegressor

scikit-learn 在 sklean.tree 包中实现了 CART 模型，分别用 DecisionTreeClassifier 和 DecisionTreeRegressor 实现了分类树和回归树。它们的训练与预测调用与其他模型类似，下面是本节开始建模时图 3-14 中所示数据的训练代码。

```
>>>from sklearn import tree                              # 引入 tree 包

>>>X = [[20、30000、400],                                # 定义训练数据
   [37、13000、0],
   …]
>>>Y = [1、0、…]

>>>clf = tree.DecisionTreeClassifier(criterion="entropy") # 初始化用熵模型训练的对象
>>>clf = clf.fit(X、Y)                                   # 训练
>>>clf.predict([[40、6000、0]])                          # 预测
[0]
>>>clf. feature_importances_                             # 查看特征重要性
[ 0.5  0.5  0. ]
```

上述代码最后访问了一个决策树模型独特的属性 feature_importances_，它返回的是模型中特征向量每个维度的重要性，数值越高重要性越大，计算重要性的依据就是该特征所在树结点的信息增益。此外，一些模型特有的重要初始化参数如下。

◎ criterion：取值 gini 或 entropy，即使用基尼指数还是熵计算信息增益。

◎ max_depth：树的最大深度。

◎ min_samples_split：继续分裂一个结点最少需要的训练样本数。

◎ min_samples_leaf：叶子结点最少含有的训练样本数。

◎ max_leaf_nodes：叶子结点的最大总数。

◎ min_impurity_decrease：用于计算叶子结点的最小纯度（即不同目标值样本之间的比值）。

这些参数主要围绕如何控制分裂结点特征的选取和防止过度拟合的训练后剪枝。

3.6.4 树的可视化

在实际应用中选择决策树模型的最主要原因就是模型易于理解，可以快速分析各特征维度的重要程度，所以 scikit-learn 为该模型提供了方便的树图导出方案。该方案在使用之前必须安装跨平台的图形库 graphviz，比如在 Mac 中可以这样做：

```
$brew install graphviz
```

在 Linux 中也可以用类似的包管理工具安装，在 Windows 中安装时可以从官网 www.graphviz.org 下载最新的安装程序。

现在可以用 sklearn.tree 包中的 export_graphviz()函数生成树图对象了，继续上一节的代码：

```
import graphviz
dot_data = tree.export_graphviz(clf, out_file=None,
                                feature_names=[u"年龄", u"收入", u"存款"],
                                class_names=[u"普通", u"VIP"],
                                filled=True, rotate=True)
graph = graphviz.Source(dot_data)
graph.render("mytree")                              # 保存成文件
```

其中参数 feature_names 用于输入特征的名称，class_names 输入目标分类名称。现在在文件系统当前目录中可以找到新文件 mytree.pdf，打开后如图 3-15 所示。

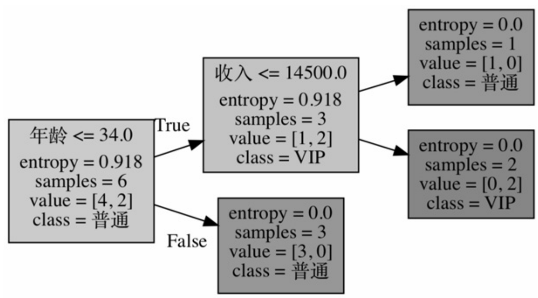


图 3-15　用 graphviz 导出树图

由于指定了 filled 和 rotate 参数，所以图 3-15 是一棵水平展开的彩色版树图。其中的颜色绘制依据是不同目标类的叶子结点有不同的颜色：本图中“class=普通”的结点为黄色；“class=VIP”的结点为蓝色；而中间结点的颜色则是其左右结点的综合。

3.7　集成学习

集成学习（Ensemble Learning）模型与其他有监督模型的出发点大相径庭，之前的模型都是在给定的训练集上通过构建越来越强大的算法进行数据拟合。而集成学习着重于在训练集上做文章：将训练集划分为各种子集或权重变换后用较弱的基模型拟合，然后综合若干个基模型的预测作为最终整体结果。

在 scikit-learn 中实现了两种类型的集成学习算法，一种是 Bagging Method，另一种是 Boosting Method。这两类方法的区别如下。

◎ Boosting Method：若干个基模型在若干个训练子集上进行互相独立的分别训练，在预测时一次性综合这些基模型的结果。

◎ Boosting Method：按迭代的顺序逐个训练基模型，在每次训练后都进行模型测试，然后根据测试结果调整下一轮基模型训练时所采用的训练数据，最后预测时仍使用所有子模型的预测得到最终结果。

本节先通过解释机器学习场景中的普遍问题——偏差与方差的原理来说明集成学习这种“分治法”能够奏效的原因，然后介绍两类集成模型中的典型算法：随机森林和自适应增强。

3.7.1 偏差与方差

现在已经掌握了多种机器学习模型的原理和调试技巧，读者是否会隐约发现每个模型虽然有各自不同的算法和若干超参数，但它们的目标惊人地相似——最后总是会集中在如何更好地适配数据，同时避免过度拟合。那么这些模型内在又有怎样的联系以致这样殊途同归呢？

1. 两类错误

这是因为有监督学习问题可以归结为对两种系统错误的最小化问题——偏差（bias）和方差（variance）。假设有一个已训练好的模型 $\widehat{f}()$，将其运行在测试数据集后，可以得到很多个错误值 $\text{Error} = \hat{f}(x) - y$，其中 x 是测试数据特征，y 是测试数据真实目标值。如果用高斯分布拟合这些错误值，最终能找到参数 u 和 σ^2，使得 $\text{Error} \sim N(u, \sigma^2)$，其中错误期望参数 u 可以理解为所谓的偏差（bias），σ^2 是方差（variance）。

说明：关于两种错误最形象的解释来自于 Scott Fortmann-Roe，有兴趣的读者可以参考 http://scott.fortmann-roe.com/docs/BiasVariance.html。

2. 与拟合程度的关系

如果分析两种错误在有监督学习预测效果上的表现，则可以发现偏差对应的是拟合不足（Underfitting）的问题，而方差对应的是过度拟合（Overfitting）的问题。正如读者一直

所看到的，拟合不足与过度拟合是两个矛盾的表征，通常满足一个就会导致另外一个出问题，因此偏差与方差也似乎总是鱼与熊掌不可兼得。在训练到一定程度后，之前学习不同模型的各种超参数通常只是用于调节偏差与方差之间的平衡点，再也无法继续缩小两者的和。

3. 集成学习意图

集成学习是一个将数据集划分为不同部分，然后各自通过能力较弱的模型进行训练，在预测时综合各弱模型结果（或投票、或取平均等）进行预测的框架。如果说之前的各种模型是一个强大独裁者，那么集成学习就是一种民主的委员会机制。

为什么这种三个臭皮匠胜于诸葛亮的模式能够成功呢？仍然用高斯分布的错误模型举例，之前学习模型都试图一步到位地找到分布在 $\text{Error} \sim N(u,\sigma^2)$ 中的最佳偏差和方差，而集成学习可以学习到若干个这样的错误分布：

$$\begin{cases} \text{Error}_1 \sim N(u_1,\sigma_1^2) \\ \text{Error}_2 \sim N(u_2,\sigma_2^2) \\ \text{Error}_3 \sim N(u_3,\sigma_3^2) \\ \qquad \cdots \end{cases}$$

这些弱模型通常有拟合不足的问题，也就是有绝对值较高的偏差、较低/一般的方差。而在预测过程综合各模型结果的过程可以看成这些错误高斯分布求平均的过程，根据高斯分布的计算公式有：$\text{Error} \sim N\left(\dfrac{u_1+u_2+\dots u_n}{N},\dfrac{\sigma_1^2+\sigma_2^2+\dots\sigma_n^2}{N}\right)$，其中 N 是弱模型的数量。

因为偏差的符号有正有负，所以若干弱模型的偏差平均可以使结果偏差从各自较高的绝对值降低到最小的程度；而对于方差来说，弱模型的方差本就不高，在综合求平均后仍能维持在较低的水平。

因此集成学习达到了能自动找到最优错误偏差和方差的效果，这对于之前的强模型来说需要数据研究者反复地调试超参数才能获得。

3.7.2 随机森林

随机森林（Random Forrest）是 Boosting Method 的一个典型代表，由它的名称就可以联想到它是一种使用决策树作为基模型的集成学习方法。

1. 集成框架

随机森林在训练过程中对训练集进行随机抽样，分别进行训练后形成若干个小的决策树。分类问题的预测通过这些基决策树的投票完成，回归问题的预测通过对基决策树结果求平均完成，整个集成框架如图 3-16 所示。

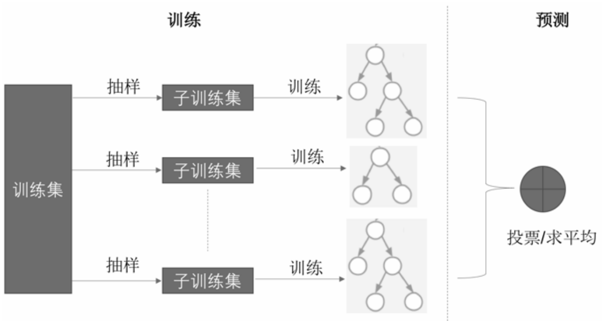


图 3-16 随机森林算法集成框架

随机森林基模型中的决策树一般采用有较大偏差和较小方差的“弱模型”，和普通的决策树相比具体体现在如下方面。

◎ 样本裁剪：通过随机采样每个弱模型只训练部分样本数据。

◎ 特征裁剪：每个基模型的决策树只选用数据特征中的一部分进行训练和预测，随机抽样保证了所有特征都能被部分弱模型学习到。

◎ 小树：由于特征和样本数量有限每个弱模型决策树都长不高，所以不需要像普通决策树那样在训练结束后为避免过度拟合而执行剪枝。

2. 有放回采样

在从整体训练集中进行随机采样划分子训练集时，随机森林通常采用所谓的有放回采样（bootstrap）手段。有放回采样是指每次抽取一个样本放入子集后将该样本仍保留在被采样空间中，使得它仍有可能被再次采样到。比如有一盘水果：1 个橘子，2 个苹果，3 个香蕉，对该盘水果进行两次有放回采样完全可能采样到 2 个橘子（虽然概率比较小）；如果使用无放回采样则最多采样到 1 个橘子。

在整体样本数量足够多的情况下，通过 bootstrap 采样到的子集只有期望约为 63.2%的无重复样本。使用这种采样有什么好处呢？

因为 bootstrap 能够生成与训练样本整体不同的数据分布，这样等于扩充了训练样本空间，所以通过 bootstrap 采样能够训练出适应性更强的模型。

3. Out-of-Bag Estimation

在第 1 章中介绍过，一般机器学习模型的评估需要通过与训练集相互独立的另一个测试集来完成。而 Bagging Method 类模型在训练过程中划分不同独立数据子集的行为在侧面引入了另一个好处，就是可以使用训练数据本身进行模型准确率的评估。

这就是 Out-of-Bag Estimation（简称 OOB），它是指基模型的评估预测只采用未参与到其本身训练的数据集。因此虽然整体上没有给出独立的测试集，但每个基模型使用的训练数据和预测数据是完全隔离的，如图 3-17 所示。

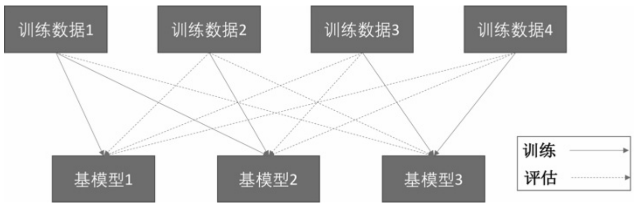


图 3-17　Out-of-Bag Estimation 示意图

图中的实线表示基模型训练时使用的数据，虚线表示进行评估预测时基模型使用的数据。只要保证每条训练数据没有被重复应用在单个基模型上，就可以保证评估结果的公允性。

4. RandomForestClassifier 和 RandomForestRegressor

在 sklearn.ensemble 包中提供了完整封装的随机森林模型 RandomForestClassifier 和 RandomForestRegressor，它们的使用方式与其他模型无异。在模型参数方面，由于随机森林使用决策树作为基模型，所以在模型初始化过程中保留了所有 DecitionTreeXXX 模型中的决策树相关参数，此外还提供了关于抽样方式和 OOB 评估的配置属性，比如：

```
    >>>from sklearn.datasets import load_iris
    >>>from sklearn.ensemble import RandomForestClassifier
    >>>iris = load_iris()                                    # 导入 iris 数据库

    >>>clf  =  RandomForestClassifier(n_estimators  =  20,  bootstrap=True,
oob_score=True)
```

```
>>>clf.fit(iris.data, iris.target)                    # 训练

>>>clf.oob_score_                                     # 查看 OOB 评估结果
0.946666666667
```

上述代码使用了 scikit-learn 的测试数据库 iris 作为训练数据来源，在初始化随机森林回归模型的时候配置 n_estimators 定义基模型的数量、配置了参数 bootstrap=True 使用有放回采、配置 oob_score=True 指定在训练后进行 OOB 测试，在训练后马上通过 oob_score_ 属性获得了准确性评估。

注意：如果未在模型初始化时指定 oob_score=True 则不能在训练后访问 oob_score_属性。

3.7.3 自适应增强

AdaBoost 是 Boosting Method 类集成算法的典型代表，其全称是 Adaptive Boosting，即自适应增强。它与 Bagging Method 类算法不同的是：Boosting Method 类算法不是通过随机抽样产生每个基模型的训练集，而是通过调整训练集中每个样本的权重使得每次迭代在不同的训练集上运行。其原理如图 3-18 所示。

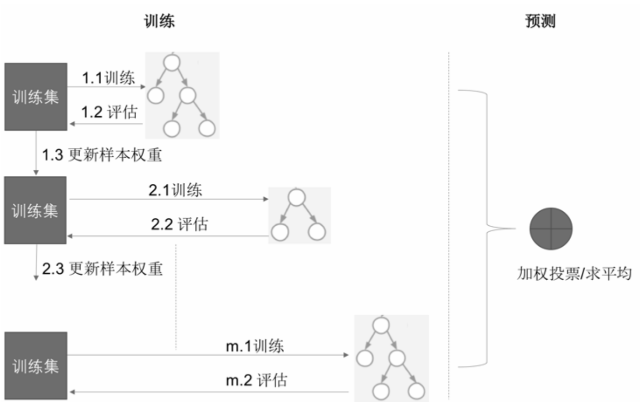


图 3-18　AdaBoost 算法原理

注意：图中基模型用决策树表示，其实也可以替换为其他模型。

1. 算法原理

在 AdaBoost 的每次迭代中都使用了全部训练样本，但训练集中的每个样本都被赋予了权值，因此基模型必须支持基于样本权值的训练方法。

注意：基于样本权值训练是指建立模型时更关注匹配权值高的样本，常见算法如支持向量机、朴素贝叶斯、决策树等都支持基于样本权值进行训练。

在第一次迭代时训练集中的每个样本会被赋予相同的权值，当训练和评估完成后，AdaBoost 会更新训练集中的样本，使得在本轮集模型评估中被预测正确的样本权值被减少、被预测错误的样本权值被增加，然后开始第二轮迭代。如此往复，每个迭代的基模型会逐渐地更关注被预测错误的那些样本，如此提高了整体的拟合效果。

虽然在最终预测时也综合全部基模型的结果，但因为基模型是随着迭代增加逐步进化的，因此在最终决策后生成的基模型往往比先生成基模型的决策权更大，所以与 Bagging Method 不同的是，最终的预测采用基模型间的“加权”投票或平均。

此外，由于每个迭代使用全部样本预测，所以 Boosting Method 类方法失去了 Bagging Method 类集成方法在训练同时进行评估的 Out-of-Bag Estimation 功能。

2. AdaBoostClassifier 和 AdaBoostRegressor

在 sklearn.ensemble 中的 AdaBoostClassifier 和 AdaBoostRegressor 分别用于 AdaBoost 的分类和回归问题。它们的训练与预测方法沿用 scikit-learn 惯例，几个独特的模型初始化参数介绍如下。

◎ base_estimator：基模型，默认为决策树，也可定义为朴素贝叶斯等模型。

◎ n_estimators：迭代次数，默认为 50。

◎ learning_rate：学习率，取值 0～1，用于控制最终决策时各基模型的权重，该值越大后产生的基模型所占的权重越大。

◎ algorithm：提升算法，可选择 SAMME 或 SAMME.R，该算法用于定义在每次迭代后如何更新训练集样本权重。SAMME 按照分类错误率更新权重，SAMME.R 按照预测概率更新权重。

由于 SAMME 只适用于离散问题，因此 algorithm 只存在于 AdaBoostClassifier 的初始化参数中，而 AdaBoostRegressor 固定使用 SAMME.R 算法。

3. 评估函数 score()

AdaBoost 不像 RandomForrest 有 OOB 评估，只能使用测试集在训练后评估。其实

scikit-learn 对大多数有监督学习模型都提供了 score()函数，用于评估测试集的预测准确率（accuracy）。使用时传入测试数据即可获得一个<=1 的评估值，该值越接近 1 表示准确率越高。结合 AdaBoostClassifier 的训练，示例代码如下：

```
>>>from sklearn.datasets import load_iris
>>>from sklearn.ensemble import AdaBoostClassifier     # 导入模型类
>>>iris = load_iris()                                  # 加载 iris 数据库

>>>from sklearn.utils import shuffle
>>>X, Y = shuffle(iris.data, iris.target)              # 对数据进行随机洗牌

#初始化以 GaussianNB 作为基模型的 AdaBoost 分类器
>>>from sklearn.naive_bayes import GaussianNB
>>>clf = AdaBoostClassifier(GaussianNB())

>>>clf.fit(X[:-20], Y[:-20])                           # 训练

>>>clf.score(X[-20:], Y[-20:])                         # 评估
0.9
```

上述代码在对 iris 数据库随机洗牌后，使用后面 20 条数据作为测试集，把前面的其他数据作为训练集，最后得到的准确度为 0.9。

注意：对于回归问题来说 score()函数的返回值不是百分比，甚至有可能为负值，所以不要误以为 0.9 一定意味着有 90%的样本正确。具体计算法则可参考 scikit-learn 官网。

在 scikit-learn 中实现了若干其他 ensemble 模型，包括 GradientBoosting、Voting 等，篇幅原因不再逐个解释。

思考：继承学习算法有哪两大类，它们的区别是什么？

3.8 综合话题

至此本章已经描述了当前最主流的几个有监督学习算法模型，本节介绍一些它们的共同话题和工具。

3.8.1　参数与非参数学习

参数学习是统计和数据挖掘领域的常见术语，在机器学习中也是一种划分算法类型的方式，出于扩充知识面的目的，这里对其进行简要介绍。

总体来说，机器学习任务可以看成对数据特征与目标值之间的映射函数的拟合：

$$y = f(x)$$

学习该假设函数有两种策略，一种是在训练之前就假设该函数的形式，学习的目的就是最优化该函数中的参数，这就是所谓的参数学习；而如果在训练之前无须定义函数形式，由数据本身去寻找可能的映射关系，这种模型则被称为非参数学习。

比如线性回归就是典型的参数学习方法，因为它在学习之前就假定映射函数符合公式 $\boldsymbol{y} = w_0 + w_1\boldsymbol{x}_1 + w_2\boldsymbol{x}_2 + w_3\boldsymbol{x}_3 + \ldots + w_{\boldsymbol{n}}\boldsymbol{x}_{\boldsymbol{n}}$，训练过程是为了寻找合适的参数 $w_0, w_1, w_2, w_3 \ldots$ 而朴素贝叶斯则是典型的非参数学习方法，因为在训练之前不假定 $y = f(x)$ 符合任何公式。

一般来说，由于参数模型事先定义了知识的形式，因此训练中对计算资源的要求会低于非参数模型，因此适用于资源有限或数据量特别巨大的场景。而非参数模型则具有更灵活的特点，在找到合适的超参数后对数据的拟合情况会好于参数学习模型。

机器学习发展至今，除线性模型、二次模型和特定领域的特殊函数模型外，几乎所有的其他模型都是非参数模型，包括朴素贝叶斯、高斯过程、决策树、集成学习等一系列算法。

注意：参数与非参数学习的概念同样适用于无监督学习，后续章节的多数模型也是非参数学习。

3.8.2　One-Vs-All 与 One-Vs-One

在 scikit-learn 中所有分类模型都支持多目标类型预测，也就是说训练数据中的目标值不只两种，还可以是更多。比如一个水果分类器不仅可以预测“是不是苹果”这样的问题，还可以预测“是苹果、是香蕉、还是橘子”这样的问题。

有些模型可以天然地支持这种多目标类型预测，比如在朴素贝叶斯模型中只要分别计算每种标签的后验概率然后进行比较即可。而有些模型则缺乏这样的能力，比如 SVM 通过超平面进行分类，其实只能区分两种目标类。那么 scikit-learn 是如何将像 SVM 这样的算法也封装成具有多目标类型预测能力模型的呢？

1. One-Vs-All

有两种机制能够完成这种将二值分类器转换为多值分类器的目的，One-Vs-All 是其中之一，有时这种策略也被称为 One-Vs-Rest。该策略为每一个目标值创建一个自己的分类器，用来分类“是不是该目标值”，在预测阶段采用最强烈地回答为“是”的那个分类器的结果。如图 3-19 所示是一个对水果进行分类的 One-Vs-All 示意图。

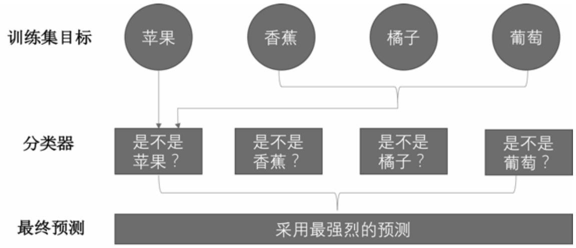


图 3-19　对水果进行分类的 One-Vs-All 示意图

图中的目标值有四种类型，所以需要建立四个二值分类器。清晰起见，图中只标注了一个苹果分类器的训练集来源，另三个分类器同理。关于“最强烈的预测”对于不同的分类器类型可能有不同的解释。比如在 SVM 中可以使用特征向量到分类超平面的距离，而在高斯过程中可以用预测方差。

2. One-Vs-One

用二值分类器实现多目标值分类的另一种方法称为 One-Vs-One，它针对每对目标值类型建立一个分类器，如图 3-20 所示。

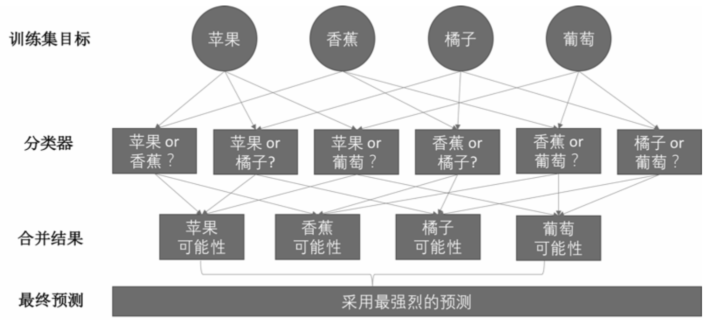


图 3-20　对水果进行分类的 One-Vs-One 示意图

在 One-Vs-One 策略中，每个分类器只用两种目标类型的数据做训练。在预测阶段，

需要先针对每个目标类型收集相关分类器的结果，再选择最强烈的预测。

3. 如何选择

scikit-learn 对所有原生二值分类器提供了 multi_class 参数用以允许设置使用如上哪种策略进行多目标值分类。两种策略在分类最终效果上差别不大，一般只需从运算性能上进行考虑。

假设训练集中共有 m 个目标值类型，则 One-Vs-All 策略需要训练 m 个分类器，而根据排列组合公式 One-Vs-One 策略需要训练 $m\times(m-1)/2$ 个分类器，因此 One-Vs-All 通常在性能上优于 One-Vs-One。

然而 One-Vs-All 的每个分类器在训练时需要用到所有数据，而 One-Vs-One 策略中的每个分类器则只需要与其相关的两个目标值类型的训练数据，因此对于一些在性能上对样本数量敏感的模型来说，One-Vs-One 的整体效率反而要高于 One-Vs-All。

3.8.3　评估工具

机器学习工程是一个不断训练新模型与评估的过程，除了每个模型基本的 score()函数可以用于评估，scikit-learn 还提供了丰富的其他工具用于模型验证。本节介绍几个相互独立的验证与评估手段，更详细的方法可参阅官网。

注意：继续阅读本节前，请读者回顾第 1 章中介绍的关于评估的知识，了解随机采样、分层采样、交叉验证、precision、recall 等概念。

1. 分割数据集

有时开发者手头只有一个数据集，但想将其分割成两部分，分别用于训练与测试。通常不能使用普通的数组分割进行这项操作，因为需要考虑样本的随机化、分层等需求。函数 sklearn.model_selection.train_test_split()提供了这些功能，比如：

```
>>>from sklearn.datasets import load_iris
>>>iris = load_iris()

>>>from sklearn.model_selection import train_test_split        # 引入函数
>>>X_train、X_test、y_train、y_test = train_test_split(
        iris.data、iris.target,                           # 传入数据整体
        train_size=0.3,                                   # 训练集的比例
```

```
        random_state=1,                                   # 随机种子
    )
```

由于 train_test_split 自动进行分层（stratified）分割，只需一行代码就可以将数据整体按比例划分为测试集和数据集。

2. 交叉验证

由于 scikit-learn 中各种算法模型都以类对象的形式封装，所以提供了模型对象和数据集，可以很方便地进行交叉验证，比如：

```
>>>from sklearn.ensemble import AdaBoostClassifier
>>>from sklearn.model_selection import cross_val_score

>>>clf = AdaBoostClassifier()                                # 模型对象
# 进行 K-fold=3 的交叉验证
>>>scores = cross_val_score(clf、iris.data、iris.target、cv=3)
>>>print(scores)
[ 0.98039216  0.94117647  0.97916667]
>>>print("total accuracy:"、scores.mean())                  # 交叉验证平均值
total accuracy: 0.966911764706
```

3. 交叉数据分割

有时需要做更加个性化的交叉验证，将数据划分与验证分开执行，因此 scikit-learn 提供了多种灵活的交叉数据分割方式，比如 model_selection.LeaveOneOut 可以生成每个测试集只有一个样本的交叉验证数据：

```
>>>from sklearn.model_selection import LeaveOneOut     # 引入模型
>>>X = [['a']、['b']、['c']、['d']]                     # 数据总体
>>>Y = ['A'、'B'、'C'、'D']
>>>loo = LeaveOneOut()
>>>for idx_train、idx_test in loo.split(X):             # 交叉数据分割
print("%s %s" % (idx_train、idx_test))

[1 2 3] [0]                                             # 分割结果
[0 2 3] [1]
[0 1 3] [2]
[0 1 2] [3]
```

注意上述代码中 split(X)函数的输出是样本索引，不是样本特征本身。因为共有四个

样本，所以生成了四个训练与预测对，其中每对的测试样本只有一个。用这样的数据集再进行交叉验证实际上能够在训练时利用更多的样本，因此在样本数量非常少的情况下比直接用 N-fold 的形式更能反映真实结果。

与 LeaveOneOut 作用类似的还有 LeavePOut（测试集中只保留 *P* 个样本，*P* 的数量可以配置）、StratifiedKFold（分层的 K-Fold 划分）、GroupKFold（某些 group 的数据用于训练，另一些用于验证）等，不再详述。

4. scoring 参数

将 cross_val_score()和其他一些 scikit-learn 验证函数在默认情况下的使用准确率（accuracy）进行验证，但如第 1 章中所描述，有时准确率不足以衡量模型的优劣。这些验证函数通常有一个参数 scoring，通过它可以配置用什么标准进行验证。利用下面一行代码可以通过错误提示的方式查看 scikit-learn 支持的所有打分（score）标准：

```
    >>>cross_val_score(clf、[]、[]、scoring='wrong_choice')

    ValueError: 'wrong_choice' is not a valid scoring value. Valid options are
['accuracy'、'adjusted_mutual_info_score'、'adjusted_rand_score'、'average_
precision'、'completeness_score'、'explained_variance'、'f1'、'f1_macro'、
'f1_micro'、'f1_samples'、'f1_weighted'、'fowlkes_mallows_score'、'homogeneity_
score'、'mutual_info_score'、'neg_log_loss'、'neg_mean_absolute_error'、
'neg_mean_squared_error'、'neg_mean_squared_log_error'、'neg_median_absolute_
error'、'normalized_mutual_info_score'、'precision'、'precision_macro'、
'precision_micro'、'precision_samples'、'precision_weighted'、'r2'、'recall'、
'recall_macro'、'recall_micro'、'recall_samples'、'recall_weighted'、'roc_auc'、
'v_measure_score']
```

这些指标可以分成三大类：

◎　accuracy、precision_XXX、recall_XXX、f1_XXX、roc_auc 等分类器指标。

◎　explained_variance、neg_mean_XXX、r2 等回归器指标。

◎　其他是聚类问题指标。

3.8.4　超参数调试

读者已经看到，为了训练好能够适配数据的模型，几乎所有的机器学习算法都有若干个超参数需要配置，这些参数传统上只能由数据研究者反复训练、记录评估、比较评估后

才能得出。现在一些机器学习开发包提供了自动化该过程的功能，sklearn.model_selection 工具包中的 GridSearchCV 类就实现了这一功能。

GridSearchCV 可以在调用者指定的各种超参数之间进行排列组合，对每一种超参数组合情况进行自动训练和评估，最后可以给出交叉验证效果最好的超参数。示例如下：

```
from sklearn import datasets
from sklearn.model_selection import GridSearchCV   # 引入 GridSearchCV
from sklearn.svm import SVC                        # 引入被调试的算法类 SVC

iris = datasets.load_iris()                        # 数据集

# 待调试超参数列表
tuned_parameters = [{'kernel': ['rbf']、'gamma': [1e-3、1e-4],
                     'C': [1、10、100、1000]},
                    {'kernel': ['linear']、'C': [1、10、100、1000]}]

######初始化 GridSearchCV，传入待调试模型、超参数列表、验证 N-Fold 值、验证标准######
clf = GridSearchCV(estimator=SVC()、param_grid=tuned_parameters
                    、cv=10、scoring='accuracy')
clf.fit(iris.data、iris.target)                    # 自动寻找超参数

###################打印自动寻找后的结果#################
print("Best parameters set found on development set:")
print()
print(clf.best_params_)                            # 最佳超参数
print()
print("Grid scores on development set:")
print()
means = clf.cv_results_['mean_test_score']         # 评价分值
stds = clf.cv_results_['std_test_score']           # 分值标准差
durations = clf.cv_results_['mean_fit_time']       # 训练时间
for mean、std、duration、params in
  zip(means、stds、durations、clf.cv_results_['params']):
    print("%0.3f (+/-%0.03f) for %r in %f seconds"
          % (mean、std * 2、params、duration))
print()
```

如代码中所示，GridSearchCV 最重要的四个初始化参数分别如下。

◎ estimator：被调试模型对象，可以是任何 scikit-learn 中的模型。

◎ param_grid：超参数的搜索空间列表，该列表中可以包含若干个 dict。每个 dict

中的 key 是超参数名，Value 是待搜索的值列表。

◎　cv：交叉验证的 N-fold 值。

◎　scoring：交叉验证使用的指标，本例是多值类型分类问题，所以采用“accuracy”。

GridSearchCV 类的 fit()方法行为与普通模型函数略有不同，它不是简单地完成训练功能，而是循环地针对每种超参数组合完成训练、交叉验证。在完成后可以读取 best_params_、cv_results_等参数查看结果，具体代码如下：

```
    Best parameters set found on development set:

    {'C': 100、'gamma': 0.001、'kernel': 'rbf'}

    Grid scores on development set:

    0.907 (+/-0.122) for {'C': 1、'gamma': 0.001、'kernel': 'rbf'} in 0.000983
seconds
    0.907 (+/-0.122) for {'C': 1、'gamma': 0.0001、'kernel': 'rbf'} in 0.001068
seconds
    0.933 (+/-0.103) for {'C': 10、'gamma': 0.001、'kernel': 'rbf'} in 0.000646
seconds
    0.907 (+/-0.122) for {'C': 10、'gamma': 0.0001、'kernel': 'rbf'} in 0.000936
seconds
    0.980 (+/-0.061) for {'C': 100、'gamma': 0.001、'kernel': 'rbf'} in 0.000501
seconds
    0.933 (+/-0.103) for {'C': 100、'gamma': 0.0001、'kernel': 'rbf'} in 0.000626
seconds
    0.980 (+/-0.085) for {'C': 1000、'gamma': 0.001、'kernel': 'rbf'} in 0.000430
seconds
    0.980 (+/-0.061) for {'C': 1000、'gamma': 0.0001、'kernel': 'rbf'} in 0.000534
seconds
    0.973 (+/-0.088) for {'C': 1、'kernel': 'linear'} in 0.000440 seconds
    0.980 (+/-0.085) for {'C': 10、'kernel': 'linear'} in 0.000462 seconds
    0.973 (+/-0.088) for {'C': 100、'kernel': 'linear'} in 0.000481 seconds
    0.980 (+/-0.085) for {'C': 1000、'kernel': 'linear'} in 0.000723 seconds
```

这样很容易就定位到了 SVC 在 iris 数据集上的最佳分类超参数——{'C': 100, 'gamma': 0.001, 'kernel': 'rbf'}。分析所有超参数组合，值得注意的是：

◎　虽然多个组合能达到 0.980 的分值，但 GridSearchCV 选择了标准差最小的一组。

◎　总体看来线性核比径向基核需要更少的训练时间，与理论相吻合。

◎ 非线性核要达到比线性核好的效果需要合适的超参数。

说明：GridSearchCV 在 fit()后还有很多其他评估结果可以查看，感兴趣的读者可以查阅在线文档。

3.8.5 多路输出

几乎所有的基本算法模型都只有单路预测功能，比如之前举例的水果分类器具备预测“是什么水果”的功能，如果还想通过它预测“有没有核”这个问题怎么办呢？虽然所有的苹果都是有核的，所有的香蕉都没核，但橘子和葡萄却不一定。

在 scikit-learn 中分别提供了 MultiOutputClassifier 和 MultiOutputRegressor 来解决这类问题。其原理是在执行 fit()函数时针对每一路输出建立单独的分类/回归器，而在执行 predict()函数时分别预测并合并结果。因此在使用方式上 MultiOutputClassifier/MultiOutputRegressor 的目标值变量须由一维数组变为二维数组，其他与普通模型无异，比如：

```
>>>from sklearn.ensemble import RandomForestClassifier
f>>>rom sklearn.multioutput import MultiOutputClassifier

>>>X = [[0、1、3]、[9、2、5]、[5、1、9]、[6、6、1]]
>>>Y = [[0、0]、[1、0]、[1、0]、[1、1]]                # 目标值是二维

# 用基模型初始化 MultiOutputClassifier
>>>clf = MultiOutputClassifier(RandomForestClassifier())
>>>clf.fit(X、Y)                                        # 训练

>>>clf.predict([[6、1、3]])                             # 预测
[[1 0]]                                                 # 预测结果值也是二维
```

只要在 MultiOutputClassifier/MultiOutputRegressor 初始化时传入基模型对象，就可以像普通模型一样进行训练与预测。

3.9 本章内容回顾

◎ 基本线性模型 OLS 的原理、实践与不足。

◎ Ridge 回归可以有效控制线性模型参数随着特征维度增加而无限增大。

◎ Lasso 回归可以将线性模型的某些参数降为零。

◎ 梯度下降优化的基本概念：假设函数是被拟合特征与目标值的映射关系；损失函数用于衡量预测值与真实值的差距，达到评价假设函数好坏的目的。

◎ 梯度下降是一种寻找最优解的方法，但有可能只找到局部最优解。随机梯度下降比普通梯度下降计算快，并且能够降低找到局部最优解的可能性。

◎ 支持向量机通过寻找最优超平面解决有监督问题。

◎ 核函数/核机制被应用在多种机器学习模型中，其作用是使模型具备解决非线性问题的能力。

◎ 概率论基本概念：事件独立性、条件概率、联合概率、贝叶斯定理。

◎ 三种朴素贝叶斯模型：高斯、多项式、伯努利，其区别在于用不同的概率分布假定计算似然度值，适用于不同的特征值类型。

◎ 高斯过程是针对无限维高斯分布的随机过程建模，可以给出对预测值的置信度方差估计，对超参数非常敏感。

◎ 信息熵用于量化计算信息价值。

◎ 决策树是基于熵理论的易于理解的模型，为防止过度拟合，在训练中需要剪枝。

◎ 集成学习是一类分割不同的训练集，用多个弱模型进行集思广益决策的框架。

◎ RandomForrest 与 AdaBoost 分别是集成学习中两种策略 Bagging Method 和 Boosting Method 的代表。

◎ 对于原生情况下只支持二值分类算法的情况，scikit-learn 通过 One-Vs-All 或 One-Vs-One 策略实现多值分类。

◎ scikit-learn 提供了丰富的数据集分割、交叉验证工具，可以验证 accuracy、precision、recall、variance 等多种指标。

◎ GridSearchCV 用于最优超参数自动定位。

◎ 多路输出的预测问题可以通过 MultiOutputClassifier/MultiOutputRegressor 来解决。

4

第 4 章

无监督学习：聚类

在机器学习中聚类（Clustering）是一种最重要的无监督学习方法，它能完成将有多维特征的数据集划分为多个子集的任务,每组内部数据之间相较其他数据有一定的亲缘性或相似性。聚类算法仍是发展活跃的领域之一,本章学习当前最具代表性的聚类算法和开发实践,主要内容如下。

◎　K-means：一种便于入门的简洁算法，派生了诸多其他划分类的算法。

◎　近邻传播：无须设置分组数量的聚类方法。

◎　高斯混合模型：用概率模型为每条数据给出多个可能聚类分组的可能性估计。

◎　DBSCAN：可在有噪声的数据空间中发现凹数据分组的密度聚类算法。

◎　BIRCH：一种能以树的形式逐级表达组间亲缘性的层次聚类算法。

◎　几类距离算法：标量距离、向量距离、时间序列数据距离。

◎　聚类评估：基本原理与常用指标介绍。

4.1 动机

如果说有监督学习分类或回归的目标可以用简单的“预测”两个字概括，那么作为无监督学习的聚类的动机则不那么明显。聚类的目标是将数据集分组，而该结果在不同的领域可以衍生出很多有价值的应用。

既然聚类的动机仅仅是“分组”二字，那么直接按点之间相互距离划分就好了，为什么会产生种类繁多的算法呢？是不是学一种或两种算法就足够了呢？这里分析一个超市门店选址问题的应用场景，逐步引出聚类中不同的技术概念，在其他应用场景中需要数据研究者举一反三。

1. 基本目标——分组

假设某集团想要在一个新的城市开几家超市旗舰店，应该如何选址才能吸引更多的市民呢？显然作为决策者不会选择偏僻的郊区，但是 N 家门店都挤在最繁华的市中心也不是最好的选择。这时如果该集团能获得本市所有居民的住址信息，则可用 K-means 等聚类算法来解决问题：

◎　将每个居民作为一个数据点。

◎　居民住址的经度、纬度作为二维特征。

◎　对所有居民进行聚类，将居民整体划分为 n 个组。

◎　每个组代表了一个居住区域，因此每个区域内可以开一家门店。

2. 质心

接下来决策者需要思考的是：在一个区域内，门店的具体地址应该是哪里呢？直观的选择是区域地理上最中心的位置（K-means 算法可以获得分组中心），但如果该区域的人口分布并不均匀，中心位置也不一定是最佳选择。这时理想的选择应该是与该区域内所有居民平均距离最短的点，也就是 AP 近邻传播等算法中的质心。

3. 层级聚类

在初步的分组之后，很可能会产生不同的组之间成员数量差异较大的情况。因此如果采用每个区域开一家门店的策略，那么市中心区域会出现太多人口共享同一门店的情况，

可能造成该门店不堪重负。

面对此情况，可以使用的手段是用 K-means 等划分型（Partition-based Methods）算法在人口较多的区域内再次聚类以产生子区域，然后看子区域内人口数量是否适合开一家门店。如此往复，逐步划分，直到找到足够小的人口数量的子区域。

上述手段最后实际上形成了一个以整个城市为根结点、逐步划分子区域的树形层次结构，这正是很多所谓层级型（Hierarchical Methods）聚类算法的动机所在，BIRCH 是一种典型的层级聚类算法。

4. 识别噪声

人口的分布总是不均匀的，任何一个城市总是既有大部分人居住的城区，也有人口非常稀薄的郊区。一些地区人口过于分散，导致商业机构只能舍弃这些区域，如何量化地划出这些不得不放弃的区域呢？

DBSCAN 是一种可以检测出数据集中噪声点的算法。使用它可以识别出哪些区域的人口过于稀疏，以便管理者对该区域选择其他营销策略。

5. 距离衡量

如何衡量点与点之间的距离是任何聚类算法的基础，上面的讨论中假定了使用地理经纬坐标的直线距离作为点之间的距离，也就是欧几里得距离（Euclidean Distance）。但是在现代城市中人们的出行必定依赖于马路网络和公共交通工具，因此更合理地衡量城市两点之间出行距离的方式应该考虑交通道路的分布情况，这就涉及用欧几里得距离之外的方式计算城市坐标之间的距离。在多维空间中如何选择不同的衡量距离方式是独立于各类算法之外的在聚类学习中必须要考虑的问题。

4.2 K-means

与有监督的分类/回归问题相比，可以将聚类问题面对的战场看成一个只有数据特征、没有目标值的数据集。K-means 是一种出发点最直观的聚类算法，在使用中它需要人为指定需要被划分组的数量，然后它将样本向量中距离最近的一些点划分到一组中。

4.2.1　算法

1. 目标

K-means 假定在聚类的每个分组中有一个中心点，算法的目的就是找到这些中心点的合适坐标，使得所有样本向量到其分组中心点距离的平方和最小。用数学公式表达，设训练集有 n 个样本，则 K-means 的目标是最小化：

$$C = \sum_{i=1\ldots N} \text{distant}(\text{centerOf}(\boldsymbol{x}_i) - \boldsymbol{x}_i)^2$$

其中 $\boldsymbol{x}_i$ 是样本的特征向量，centerOf$(\boldsymbol{x}_i)$ 是样本所在组的中心点向量（一个样本与哪个中心点更近，则被分为与该中心点一组），distant() 用于计算两个向量之间的距离。该公式计算结果在下文中简称 C 值。

2. 步骤

假设数据集需要被划分为 M 个组，K-means 使用逐步迭代的方式找到每个中心点的合适位置。最初，可以用固定值或随机的方式选取任意 M 个点作为每组的中心点，然后迭代执行如下两步操作：

- ◎ 为每个中心点找到各自的样本数据，一个样本数据距离哪个中心点近则被划分给哪个中心点。
- ◎ 在每个组内用该组成员重新计算出中心点，新的中心点坐标是组内每个成员在各维度上的算数平均值，新中心点确定后的 C 值一定低于或等于原来的 C 值。

上述循环在到达算法收敛条件后退出。所谓收敛条件是 C 值低于某个要求或者每组中被推举的中心点不再发生变化。

注意：K-means 的中心点向量不一定是训练样本中某成员的位置。

3. 演示

出于便于理解的目的，使用二维特征向量演示用 K-means 算法逐步找到中心点合适位置的算法过程。假设平面上有如图 4-1 所示的若干个点，试图使用 K-means 算法将它们分为三组。

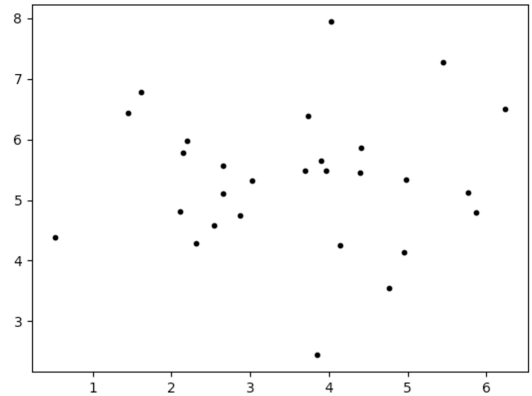


图 4-1　二维特征样本集

首先可以用随机方法选定三个中心点，然后划分样本数据到各自管辖范围内。使用三种图形和颜色分别表示三个组，第一个迭代的结果如图 4-2 所示。

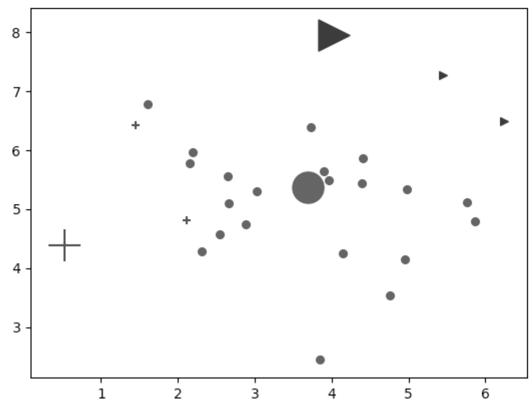


图 4-2　K-means 迭代的结果 1

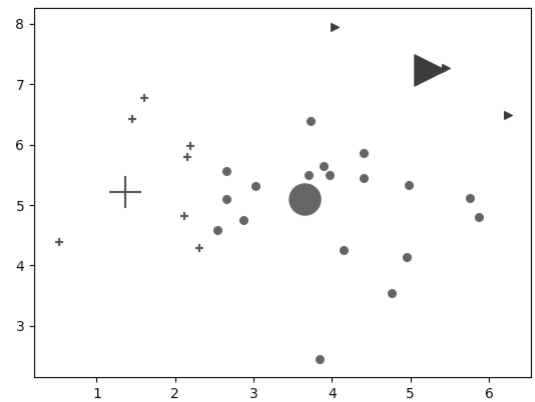


图 4-3　K-means 迭代的结果 2

图中的大散点是以随意选取的三个样本数据作为初始中心点，在选定它们之后可以将所有的训练数据样本（小散点）划分到各自组中。经过计算此时的 C 值约为 68.66。现在开始第二轮迭代，根据当前分组情况推举出新的中心点，结果如图 4-3 所示。

图 4-3 中各组的“中心点”被移到了各自组中更加中心的位置，由于中心点的变化，样本数据的分组情况也随之发生变化。如此变化后的系统 C 值也随之变小，当前 C 值约为 48.26。如此重复迭代总会到一个阶段：无法再调整中心点使 C 值变小。因此来到如图 4-4 所示的最终状态。

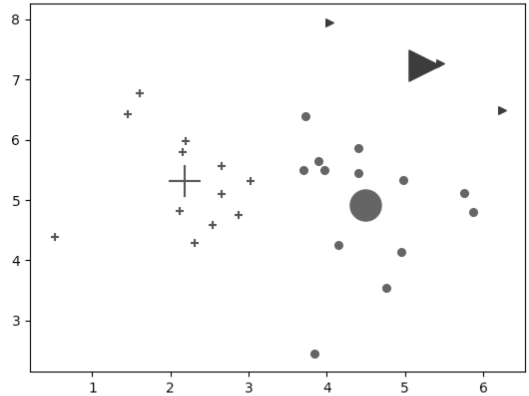


图 4-4　K-means 迭代的最终状态

此时的 C 值为 36.32，并且无法进一步减小，说明系统已经达到了收敛的稳定状态，K-means 算法到此完成。

4.2.2　实战：scikit-learn 聚类调用

在 scikit-learn 的 sklearn.cluster 包中提供了多个聚类算法的封装模型。不同的模型在调用方式上大体相同，不同之处在于各算法有不同的模型初始化参数和模型属性。本节介绍 sklearn.cluster.KMeans 的具体调用方式。

1. 训练和预测

与有监督学习的分类/回归模型一样，聚类模型最重要的调用仍然是训练函数 fit()与预测函数 predict()。由于无监督学习没有人为给出的目标值，在 scikit-learn 语境中聚类问题的预测目标值变为了“分组编号”。因此聚类训练除了计算出模型内在的预测参数，同时也是对训练数据进行预测的过程。官网上的一段示例代码如下：

```
>>>import numpy as np
>>>from sklearn.cluster import KMeans                  # 引入 K-means 模型

>>>X = np.array([[1, 2], [1, 4], [1, 0],               # 样本特征向量
            [4, 2], [4, 4], [4, 0]])

>>>kmeans = KMeans(n_clusters=2, random_state=0).fit(X)        # 训练
>>>kmeans.labels_                                       # 训练数据的聚类结果
```

```
[0 0 0 1 1 1]

>>>kmeans.predict([[0, 0], [4, 4]])                    # 预测新数据的类别
[0 1]
```

在训练后读取模型的 labels_属性可以获得训练数据的聚类结果。由于聚类算法本身对聚类后每个分组的含义一无所知，所以只能用 0、1、2……这样的数组对分组进行编号。使用新数据调用 predict()可以直接获得数据的分组编号。

2. 中心点与收敛值

在训练后可以读取模型的 cluster_centers_和 inertia_参数获得中心点坐标和最终收敛值（前文算法讲解中的 C 值）。继续上面的代码：

```
>>>kmeans.cluster_centers_
[[ 1.  2.]
 [ 4.  2.]]

>>>kmeans.inertia_
16.0
```

上述代码获得两个中心点向量<1, 2>和<4, 2>，中心点是 K-means 模型中后续执行 predict()进行样本组别划分的唯一标准。

3. 距离查询

除了直接预测特征向量属于哪个分组，scikit-learn 还提供函数 transform()用于计算特征向量距离所有中心点的距离。很显然，距离哪个中心结点更近则更可能属于哪个分组。通过比较与不同中心点的距离，这种预测方式在一定程度上提供了分组置信度，也给使用者提供了进行目标组别排序的可能。示例代码如下：

```
>>>kmeans.transform([[4, 2], [4, 4]])          #查询两个向量与中心的距离
[[ 3.          0.        ]
 [ 3.60555128  2.        ]]
```

第一个向量与中心点 1 的距离为 3，与中心点 2 的距离为 0；第二个向量与中心点 1 的距离是 3.60555128，与中心点 2 的距离是 2；显然两者都属于第二个分组，但第一个向量的确定性更高一些。

4. score()函数

在有监督学习模型中，score()函数通过比较预测目标值与真实目标值来评估模型对数据的拟合程度。而在无监督学习的聚类中，scorce()函数只能用于衡量模型对测试数据的分组信心。比如在 K-means 中，测试数据离中心点越近则 score()分值越高，反之则越低。继续如上代码：

```
>>>kmeans.score([[2, 2], [5, 3]])              #测试数据离中心点较远
-3.0

>>> kmeans.score([[1, 2], [4, 2]])             #测试数据即中心点
-0.0
```

上述模型的中心点向量是<1, 2>和<4, 2>，当测试数据都坐落在中心点上时，score()返回-0.0，离中心点越远返回一个越小的负值。

5. 初始化参数

与之前的各种 scikit-learn 模型一样，开发者可以对 K-means 的一些算法超参数以类对象初始化参数的形式进行配置。常用的 K-means 超参数如下。

- ◎ n_clusters：把整体数据集分为几个组。
- ◎ init：初始中心结点的选择策略，可以是“random”“k-means++”或一个自定义的数组向量。其中“k-means++”是一种倾向于选择相互距离较远的中心点初始化方法。
- ◎ max_iter：最多进行几次重新计算中心点的迭代。
- ◎ tol：inertia_小于多少时认为已经收敛并立即停止迭代，也就是本节算法介绍中的 C 值。
- ◎ n_init：进行几轮结点初始化和迭代。

由于 K-means 算法本身不保证全局最优解，所以 K-means 进行多轮“从 K-means 初始化到迭代收敛”的完整过程，选择其中 inertia_最小的一轮作为最终结果。

4.2.3 如何选择 *K* 值

在 K-means 算法中有一个必须人为指定的超参数 n_clusters，即定义需要将数据集划分为几组，该参数习惯上被称为“*K* 值”。在不同的场景中 *K* 值的选择没有固定的标准。

K-means 聚类结果的一个衡量标准是 inertia_值越小越好，因为该值越小说明每组的成员数据之间距离越近。但随着 *K* 值的增加，inertia_会越来越小，直到 *K* 值与样本数量相同时，inertia_变为零。在任何数据集上，*K* 值与 inertia_是反相关的关系，可以将 *K* 值的增大看成一种对 inertia_减小的贡献。但显然 *K* 值不能无限增大，因为这样就失去了聚类的意义。

那么什么是合适的 *K* 值呢？一个比较直观的方法是，选择对 inertia_贡献效果已经不显著的 *K* 值点，如图 4-5 所示。

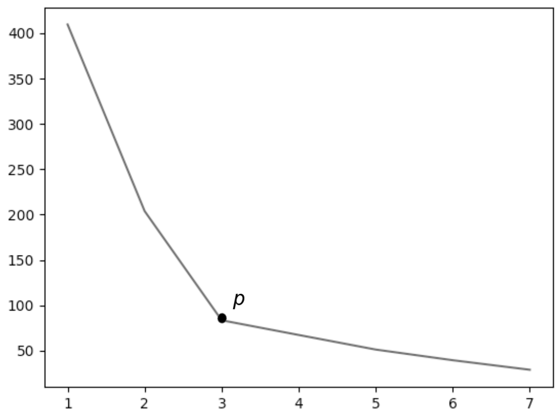


图 4-5　*K* 值与 inertia_值

图 4-5 的横坐标是 n_clusters 值（即 *K* 值），纵坐标是 inertia_值。当 n_clusters=1、2 时，每增加一个 *K* 值单位能对 inertia_产生显著的降低效果，而当 n_clusters=3 时再增加 *K* 值对 inertia_降低的效果就比较少了。因此在该数据集中，3 是一个比较稳妥的 *K* 值选择，此时的点 *P* 也称为拐点。

实际上图 4-5 是对如图 4-6 所示的数据集在不同 *K* 值下的测试。

在视觉上就可以分辨出将数据集划为三个组是一个合理的选择。当数据的特征维度较高无法用视觉识别时，采用查找 *K* 值曲线拐点的方法仍然可以找出合适的 *K* 值。

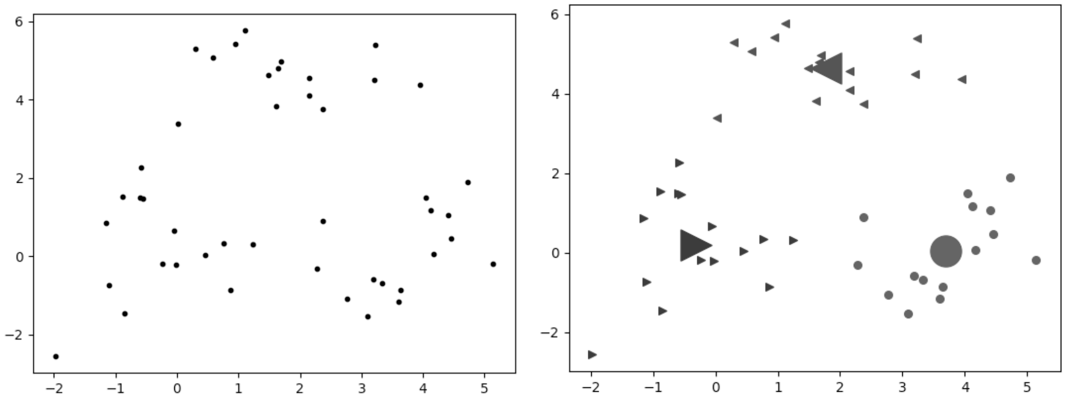


图 4-6　*K* 值测试数据集（左）和最佳 *K* 值聚类效果（右）

4.3　近邻算法

近邻算法（Affinity Propagation，AP）是 *Science* 在 2007 年发布的聚类算法，其基本目标与 K-means 一致，都是追求找出组内距离平方和最小的划分方法。但 AP 算法具有结果稳定可重现（K-means 结果在一定程度上依赖于开始的随机中心点）、训练前无须指定分组数目（K-means 中的 *K* 值）等优点，只是算法的时间复杂度较 K-means 稍高。

4.3.1　生活化的理解

要理解近邻算法需要了解若干算法本身自定义的概念，比较难以入手，但可以通过一个比较生活化的例子掌握这些理念。设想这样一个场景：某门课程的期末考试通过组建小组完成指定项目的方式进行，老师要求同学们自行分组，对每组的人数上限没有要求，但每组需要有一名组长。这时同学们会如何分组呢？

当老师宣布这个想法后，课堂内也许会立即响起同学们之间讨论的声音：

李林：王浩，我们组队吧，我选你作组长。
王浩：李梅梅学习更好，我觉得应该和她一组。
李林：好吧，我和她合不来，我还是去问问吴丽颖吧。
李梅梅：大家不要选我做组长啊！我只适合当参谋。
……

这些讨论无非是两个话题：我希望和谁一组并选谁做组长、我自己是不是想做组长，

在经过一阵子讨论后，大家总能找到合适的组织并推选出组长。

如果将同学分组看成一个聚类过程，则上面的流程恰好演绎了近邻算法的分组过程。在该过程中用到的几个 AP 概念如下。

◎ 质心（exemplar）：也就是同学组长，是聚类中每一组的核心成员。

◎ 参考度（preference）：是每一个同学想担任组长的意愿程度，在聚类之前可以对每一个成员设置一个参考度值。

◎ 相似度（similarity）：记为 $s(i, k)$，可以看成同学 i 与 k 在讨论之前的熟悉程度，该值越高则两个同学越可能分成一组。在普通聚类场景中，也就是两个特征向量之间的距离。

◎ 责任度（responsibility）：记为 $r(i, k)$，是同学 i 对同学 k 说的一句话，其内容是“我想选你做组长的意愿是多少”，当然该值越高说明 i 越想加入 k 的一组。

◎ 可用度（availability）：记为 $a(i, k)$，是同学 k 对同学 i 说的一句话，其内容是“我想当组长的意愿是多少”，该值越高说明 k 越可能成为质心。

其中质心在聚类完成之后正式产生；参考度和相似度是在聚类之前需要给出的超参数；责任度和可用度是在聚类过程中使用的算法概念。

4.3.2 有趣的迭代

现在可以开始看看近邻算法的工作原理了。与 K-means 类似，AP 也是用迭代的方式逐渐找到质心/中心点。

注意：近邻算法中的质心与 K-means 的中心点略有不同，K-means 的中心点可以是特征取值空间中的任意一个点，而质心必须是样本数据中的某个点。

每两个结点之间都有相应的 $r(i, k)$和 $a(i, k)$，因此可以将系统整体的责任度和可用度各自看成一个二维矩阵，AP 算法迭代的目标是逐步更新责任度和可用度矩阵。每个迭代分两步执行。

◎ 更新责任度矩阵：每个 $r(i, k)$新的值等于原始相似度 $s(i, k)$减去上一轮迭代 i 结点收到的最大“相似度+可用度”组合（第一轮迭代所有结点的可用度为零）。可以理解成“如果有其他结点更愿意做质心，则结点 i 发给 k 的责任度降低”。

◎　更新可用度矩阵：每个 $a(i, k)$新的值等于其自责任 $a(k, k)$加上其他结点发给 k 的所有正向责任度的和。可以理解为“有越多的结点希望 k 做质心，则其越自告奋勇地争当质心”。

每一轮迭代其实都是一个聚类结果的瞬间状态。在该状态中将责任度矩阵与可用度矩阵相加，则对于每一个结点 i 来说能够获得 $r(i, k)+a(i, k)$最大值的那个 k 结点就是 i 所在组的质心。该迭代过程的收敛停止条件可以达到最大的迭代次数，或者连续若干次迭代聚类结果没有发生变化。

仍然用图 4-1 中的数据演示 AP 算法，该数据集的迭代过程和聚类结果如图 4-7 所示。

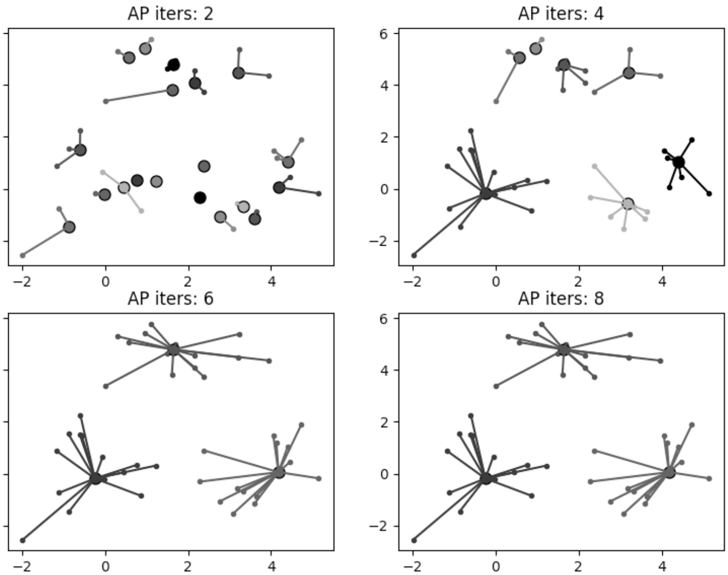


图 4-7　随着迭代变化的 AP 聚类结果

图 4-7 是在设置所有结点初始 Preference=−50 的情况下分别在迭代 2、4、6、8 时绘制的聚类结果图，图中稍大结点是每组的质心。在算法开始时所有结点各自为政，然后逐渐和附近的结点并为一组，直到分组稳定。本例在迭代 6 的时候已经达到了最好的分组状态，所以即使继续迭代到 8 分组结果仍然没有变化，此时可以认为 AP 聚类已经完成。

4.3.3　实战：AffinityPropagation 类

scikit-learn 中的 sklearn.cluster.AffinityPropagation 实现了 AP 算法的基本功能。对于

该类来说比较重要的初始化参数如下。

◎ damping：阻尼因子，范围在 0.5～1 之间，该值越大每次迭代中责任度矩阵和可用度矩阵更新越慢，因此算法收敛也越慢。但较大的 damping 可以防止这些值在更新中的抖动，降低无法收敛的风险。

◎ convergence_iter：聚类结果连续 convergence_iter 次迭代没有变化时，认为已经达到稳定状态，算法完成。

◎ max_iter：算法的最大迭代次数，到达该值时即使没有达到稳定状态也不再继续。

◎ preference：结点参考度，可以是一个数值（所有样本数据使用相同参考度），也可以是一个数组（每个样本有各自的参考度）。

◎ affinity：相似度的计算方法，可以是“euclidean”或“precomputed”。前者是默认值，指用特征向量之间的欧几里得距离计算相似度 $s(i, k)$；后者是指开发者自己计算 $s(i, k)$，此时在给 fit()、predict()等函数传递参数时需要传入相似度矩阵，而不是特征向量列表。

在训练完成后可以读取模型的下列属性获得聚类结果如下。

◎ cluster_centers_indices_：质心样本在训练集中的索引号。

◎ cluster_centers_：质心结点的特征向量数组。

◎ labels_：训练样本的聚类结果。

◎ affinity_matrix_：近邻矩阵，也就是责任度矩阵与可用度矩阵的和。

◎ n_iter_：算法收敛所用的迭代次数。

示例代码如下：

```
>>>import numpy as np
>>>X = np.array([[1, 2], [1, 4], [0.7, 0], [0.2, 5], [0, 4], [1.3, 0],
# 训练数据
            [0.1, 2], [0, 4], [0.4, 0]])

>>>from sklearn.cluster import AffinityPropagation        # 引入 AP 算法类

>>>af= AffinityPropagation(preference=-5, ).fit(X) # 用 preference=-5 聚类
>>>af.labels_                                      # 查看聚类结果
[2 1 0 1 1 0 2 1 0]
```

```
>>>af2= AffinityPropagation(preference=-8, ).fit(X)# 用 preference=-10 聚类
>>>af2.labels_
[0 1 0 1 1 0 1 1 0]
>>>af2.n_iter_
42                                                         # 迭代进行了 42 轮
>>>af2.cluster_centers_
[[ 0.7  0. ]                                               # 两个质心的坐标
 [ 0.   4. ]]
```

代码中对相同的数据集用不同的 preference 参数进行了两次聚类，获得了不同的聚类结果。第一次聚类将数据集划分成了三个组，而第二次聚类划分了两个组。用 Matplot 绘制以上代码中的数据，效果如图 4-8 所示。

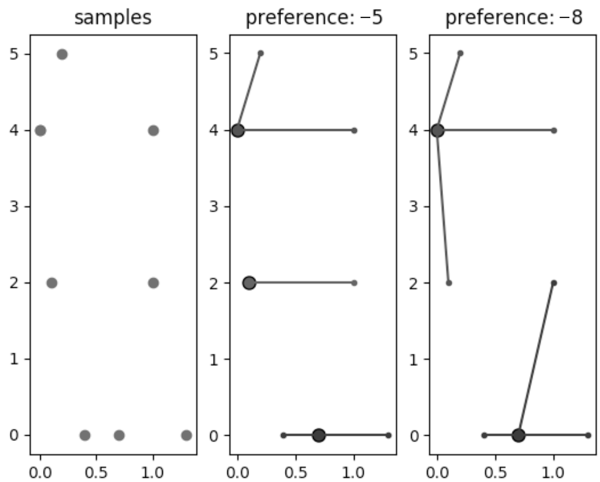


图 4-8　参数 preferenc 对 AP 聚类结果的影响

在图 4-8 中，左图是原始数据集，中图是 preference=–5 时的聚类结果，右图是 preference=–8 时的聚类结果。

4.4　高斯混合模型

本节学习一种概率聚类模型——高斯混合模型（Gaussian Mixture Model，GMM）。GMM 假设训练数据集是在若干多元高斯分布中随机采样获得的结果，因此学习的目标就是获得这些高斯分布的均值与方差。和有监督学习中的朴素贝叶斯、高斯过程等相似，概率类模

型的好处是在做出预测/分组值估计的同时给出置信概率，并且可以给出第二选择、第三选择等一系列预测。

初学者阅读本节之前建议回顾第 3 章中朴素贝叶斯、高斯过程中的概率相关内容。同时，本节讲解的高斯分布、最大似然估计、协方差矩阵类型等内容不仅适用于 GMM 模型，也是后续的隐马尔可夫模型、贝叶斯网络等章节的基础。

4.4.1 中心极限定理

在第 3 章的高斯过程一节中已经介绍了易于计算是很多概率模型选择高斯分布作为假设函数的原因之一。但现实中的数据千差万别，凭什么说高斯分布就能很好地匹配要学习的数据呢？这就不得不提概率类模型世界的核心理论——中心极限定理了。

中心极限定理（Central Limit Theorem）是数理统计学和误差分析的理论基础，指出了大量随机变量累积分布函数逐渐收敛到高斯分布的积累分布函数。其最初来源于拉普拉斯“当试验次数趋于无穷时，二项分布的极限是高斯分布”的重要证明，后来任意类型独立随机事件的极限都被证明可用高斯分布组合表示。

高斯分布的另一个名字是正态分布，可以用简短通俗的语言解释中心极限定理：正态分布无处不在。这也是学校里的老师常说班级里同学的成绩符合正态分布、人群的身高总是中值附近的人数最多、公司里的大多数员工都不显山露水“碌碌无为”的原因。

在被学习的数据中，虽然有时正态分布的特征不能一眼被发现，但当数据量足够大时往往能发现其背后的高斯分布，如图 4-9 所示的 iris 数据集。

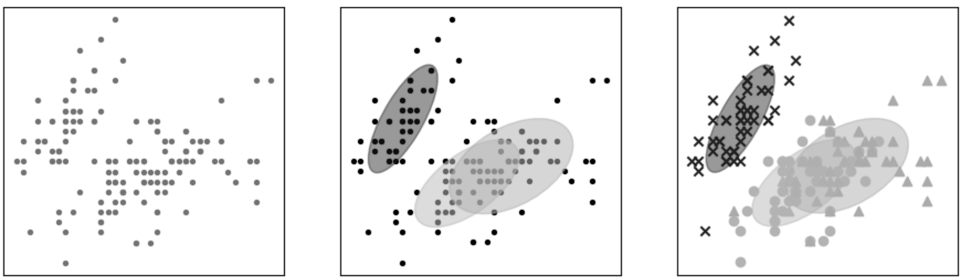


图 4-9　高斯分布拟合 iris 数据

鸢尾花卉数据 iris 是 scikit-learn 提供的一个测试数据集，其中包括三类花卉的共 150 个样本。如图 4-9（左）所示是用 iris 中所有数据的前两个维度特征绘制的散点图，通过

该图似乎找不到任何高斯分布的迹象；如图 4-9（中）所示是用高斯混合模型拟合该数据集后绘制的 3 个二元高斯分布，左上角的一个分布比较明显地表示出了一类数据，但右下角的两个重叠较多的分布是否有意义呢？如图 4-9（右）所示用不同的形状表示出了每个散点的真实类别，可以发现右下角的两个分布恰巧大致拟合了两个类别的数据。

本例也给出了概率模型相比之前的 K-means、AP 等距离模型的一个优点：有时基于已有特征确实很难区分某个数据属于哪个类别（两个分布的重叠区域），此时与其像 K-means 等模型那样粗略给出一个分组，不如像 GMM 这样给出其所属分组和相应的概率。

4.4.2　最大似然估计

已经掌握了足够多 GMM 模型动机方面的内容，那么能由样本数据拟合出高斯分布的均值和方差参数的原理是什么呢？这里 GMM 采用了统计学中非常通用的一个参数估计方法——最大似然估计。

在第 3 章朴素贝叶斯一节中已经介绍了似然度 $P(D|H)$ 的概念，在机器学习中 D 可以看成样本数据集、H 可以看成概率模型参数。最大似然估计的目标就是找到在给定样本集 D 的情况下能使似然度值最大化的一组模型参数 H（对高斯分布来说就是均值和方差）。

由于数据集 D 由很多样本组成，则 $P(D|H)$ 可以看成多个样本的联合概率。假定样本之间相互独立，设有 n 个样本，则由独立事件乘法定理有：

$$P(D|H)=\prod_{i=1}^{n}P(d_i|H)$$

对于高斯混合模型的聚类问题来说，每个样本发生的概率：

$$P\left(d_i|H\right)=\sum_{i=1}^{K}p\left(k\right)p(d_i|k)=\sum_{i=1}^{K}p\left(k\right)N(d_i|\mu_k,\Sigma_k)$$

其中 K 是聚类分组数量，$p(k)$ 是每个分组出现的概率，$N(\mu_k,\Sigma_k)$ 是 K 个多元高斯分布。由此可见似然值 $P(D|H)$ 最终取决于 $p(k)$、μ_k、Σ_k 这些参数，这样 GMM 最大似然估计的目标明确成了寻找 $p(k)$、μ_k、Σ_k 以使得 $P(D|H)$ 有最大极值。

在大学的数学与统计课程中，求极值问题有固定的“套路解法”——对目标函数求导后解方程组。虽然这种解法能够找到全局最优解，但是计算代价特别大，不太适合在大数据环境中实现。在 GMM 中采用与梯度下降、K-means 找中心点等类似的思想做最大似然

估计，也就是假设一组起点 $p(k)$、μ_k、Σ_k 参数后分两步迭代搜索修改这些参数：

◎　计算每个样本 i 在这些参数下是由高斯分布 k 产生的概率 $\gamma(i,k)$。

◎　用这些 $\gamma(i,k)$ 概率值推选出新的 $p(k)$、μ_k、Σ_k。

重复以上步骤直到参数不再变化或者达到最大迭代次数，即可得到最终 GMM 最大似然估计参数。其中利用 $\gamma(i,k)$ 进行计算的公式比较复杂，这里不再列出。

这样的迭代求解其实有一个正式名称——EM 算法。与梯度下降、K-means 等一样，用 EM 算法求解的 GMM 有可能陷入局部极值。

4.4.3　几种协方差矩阵类型

GMM 聚类最重要的求解目标是每个分组的 μ_k 和 Σ_k，其中 μ_k 是每个分组的均值点，Σ_k 是该分组高斯分布的协方差矩阵。如果数据集有 M 个特征，则协方差矩阵的形状是 $M \times M$，Σ_k 中的每个元素表示了特征之间的两两相关程度。

在机器学习环境中 M 值通常不小，也就意味着 Σ_k 中实际上包含着很多待求变量。这就产生了两个可能的问题：

◎　如果训练集中样本数量 N 很小，则可能不足以拟合这么多的协方差变量。

◎　过多的协方差变量会导致太长的训练计算时间。

因此在某些场景中适当减少 Σ_k 中的待求变量成为一种合理诉求，大多数的机器学习高斯分布相关模型通过选择协方差矩阵类型来达到这一目的。常用的协方差矩阵类型有四种：球面（spherical）、对角（diagonal）、绑定（tied）、完全（full）。

1. 球面协方差矩阵

球面是最简单的一种协方差矩阵，它将 Σ_k 退化变为单独的一个待求变量 σ，协方差矩阵的形式是 $\begin{pmatrix} \sigma^2 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & \sigma^2 \end{pmatrix}$，即除对角线上都是相同的一个元素 σ 外，其他元素都是零。对角线上的元素代表每个特征维度本身的方差，因此球面协方差矩阵的含义是:所有特征的方差都相同，并且特征之间相互独立。这种高斯分布体现在图形上就是一个球型，如图 4-10（左）所示，这也是球面协方差矩阵名称的来源。

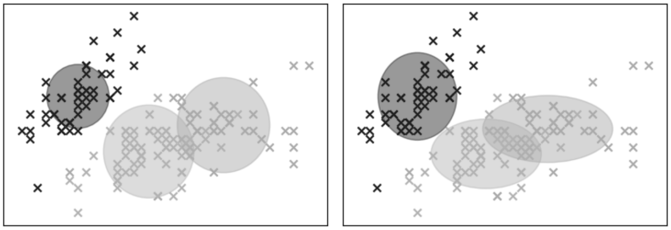


图 4-10　球面（左）与对角（右）协方差矩阵效果

2. 对角协方差矩阵

在对角协方差矩阵中，Σ_k 对角线上的每个元素都有一个待求变量 σ，其他所有元素保持为零，即 $\begin{pmatrix} \sigma_1^2 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & \sigma_m^2 \end{pmatrix}$。这样数据集上的每个维度都有自己的方差，但维度之间仍然相互独立。体现在二维图形上，每个高斯分布不再是完美的圆形，由于横/纵坐标有不同的方差，因此变为椭圆形。但是由于特征之间相互独立，所以椭圆的长轴/短轴必定平行于横/纵坐标系。不管数据分布情况如何，它们看上去都是端端正正地放在那里，如图 4-10（右）所示。

3. 完全协方差矩阵

完全协方差矩阵 Σ_k 中的每一个元素都是一个待求变量，即 $\begin{pmatrix} c_{11} & \cdots & c_{1m} \\ \vdots & & \vdots \\ c_{m1} & \cdots & c_{mm} \end{pmatrix}$。在数据足够的情况下它能表达最完整的高斯分布信息。此时不但每个特征有自己的方差值，任意两个特征之间还可以有相关性。因此图形上的椭圆不再垂直于坐标轴，而是尽量拟合数据分布情况，如图 4-11（右）所示。

4. 绑定协方差矩阵

绑定协方差矩阵中的每一个元素也都是一个待求变量，与完全矩阵不同的是，它要求 GMM 中所有高斯模型采用同一个协方差矩阵。体现在图形上则是所有高斯分布椭圆的大小与方向都完全相同，如图 4-11（左）所示。

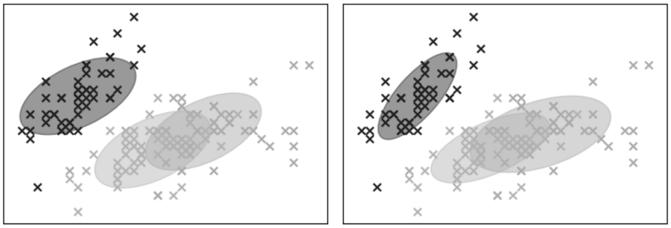


图 4-11　绑定（左）和完全（右）协方差矩阵效果

4.4.4　实战：GaussianMixture 类

scikit-learn 在 sklearn.mixture 包中提供了 GMM 模型封装类 GaussianMixture。在掌握了 GMM 算法原理后很容易理解该模型的使用方法。

1. 初始化参数

GaussianMixture 的初始化模型几乎都与高斯分布和控制最大似然估计流程有关，具体参数如下。

- ◎ n_components：聚类分组个数，与 K-means 一样，模型使用者必须提供该参数。
- ◎ covariance_type：协方差矩阵类型，可以是 spherical、diag、full 或 tied。
- ◎ tol：EM 算法收敛阈值。
- ◎ max_iter：EM 算法最大迭代次数。
- ◎ n_init：训练开始前选择 $p(k)$ 、μ_k 、Σ_k 参数的次数，对数据匹配最好的一次将被用于开始 EM 迭代。
- ◎ init_params：用什么方法初始化 $p(k)$ 、μ_k 、Σ_k 等参数，可选 K-means 或 random。
- ◎ weights_init、means_init、precisions_init：可以传入调用者自定义的初始 $p(k)$ 、μ_k 、Σ_k ，其中 Σ_k 以精度矩阵的形式给出。

其中精度矩阵（precision matrix）是协方差矩阵 Σ_k 的逆矩阵，在 GMM 中使用它是因为计算上的方便性，对其有兴趣的读者可参阅统计学教材。

2. 模型属性

在调用 fit()方法训练后可以通过下列属性获得 GMM 模型结果信息如下。

- ◎ weights_：每个聚类分组的比重。
- ◎ means_：每个高斯分布的均值点。
- ◎ covariances_：每个高斯分布的协方差矩阵，矩阵的形状取决于协方差矩阵类型。
- ◎ precisions_：精度矩阵。
- ◎ precisions_cholesky_：精度矩阵的 cholesky_分解。
- ◎ converged_：EM 算法最终是否收敛（如果由于达到 max_iter 而停止迭代，则不算收敛）。
- ◎ n_iter_：迭代次数。
- ◎ lower_bound_：最终在训练数据上达到的似然度水平，以对数方式表达。

其中需要稍作解释的是 cholesky_分解。所谓 cholesky_分解是将对称正定矩阵用下三角矩阵与其转置乘积表达的一种分解方式，即 $\boldsymbol{A}=\boldsymbol{L}\boldsymbol{L}^{\mathrm{H}}$，其中 $\boldsymbol{L}$ 是下三角矩阵，$\boldsymbol{L}^{\mathrm{H}}$ 是 $\boldsymbol{L}$ 的共轭转置（在实数矩阵下也就是转置矩阵 $\boldsymbol{L}^{\mathrm{T}}$）。分解后的矩阵 $\boldsymbol{L}$ 便于后续计算，所以 scikit-learn 为概率相关模型提供了该属性。

3. 方法

GaussianMixture 上的函数与其他模型类似。稍有不同的是，GaussianMixture 是概率模型，它实现了 predict_proba()函数用于在预测的同时给出样本在各分组高斯分布上出现的概率。举例如下：

```
>>>import numpy as np
>>>from sklearn import datasets
>>>from sklearn.mixture import GaussianMixture        #引入 GMM 模型
>>>from sklearn. model_selection import train_test_split

>>>iris = datasets.load_iris()                        # 加载数据集
>>>X_train,X_test, y_train, y_test =
     train_test_split(iris.data,iris.target,test_size=0.4)
```

```
# 初始化一个 diagonal 类型协方差矩阵的 GMM 模型，其中包含 3 个高斯分布
>>>gmm = GaussianMixture(n_components=3, covariance_type="diag",
                  max_iter=20, random_state=0)
>>>gmm.fit(X_train)                                          # 训练

# 查看训练后的模型结果
>>>gmm.n_iter_                                               # 迭代次数
2

>>>gmm.weights_                                            # 每个分组的比重
[ 0.28613186  0.37777778  0.33609036]

# 查看均值，因为 iris 数据集有四维特征，所以是一个 3×4 的矩阵
>>>gmm.means_
[[ 6.89093848  3.11866322  5.78778038  2.09770464]
 [ 4.99411765  3.38235294  1.45294118  0.22941176]
 [ 5.89277586  2.76012405  4.30287213  1.38786049]]

# 查看协方差矩阵，由于使用的 diagonal 类型数据有四维特征，所以每个矩阵只用四个数值表示
协方差矩阵对角线上的元素。
>>>gmm.covariances_
[[ 0.28767562  0.09097316  0.27983341  0.05517405]
 [ 0.12584875  0.15321899  0.02013941  0.00913595]
 [ 0.19614476  0.08245879  0.27876302  0.0726976 ]]

>>>gmm.predict(X_test[:1])                                   # 预测
[0]
>>>gmm.predict_proba(X_test[:1])                             # 概率预测
[[  9.74704632e-001   1.01299386e-254   2.52953681e-002]]
```

代码中的 predict_proba()函数给出的三个值中第一个值最大，因此该条测试数据被预测成分组 0（即第一个分组）。

4.5 密度聚类

密度聚类即 DBSCAN，全称是 Density Based Spatial Clustering of Applications with Noise，它是密度聚类算法的典型代表。与之前的算法相比，密度聚类的每个分组没有绝对的中心点，相反每个样本都可以成为其他样本的核心以扩展自己的分组。在聚类效果上，

密度算法最大的优点是能更好地聚类非凸数据集，并且不需要在训练前指定分组数量。

4.5.1　凸数据集

之前学习的 K-means、AP、GMM 等都有各自的优点，也许读者已经觉得它们能够适应绝大多数的聚类场景。但其实它们有一个共同的缺点，就是只能适配凸数据集（Convex set）。对于没有学过统计课程的读者来说，可能凸数据本身是一个模糊的概念：似曾相识却又难以表达。

其实凸数据在拓扑学上是有明确定义的：如果在一个数据集的任意两点之间作出的线段都完全在该数据集内部，则称该数据集是凸集合。如图 4-12 所示是 Wiki 上对凸集合与凹集合的形象解释。

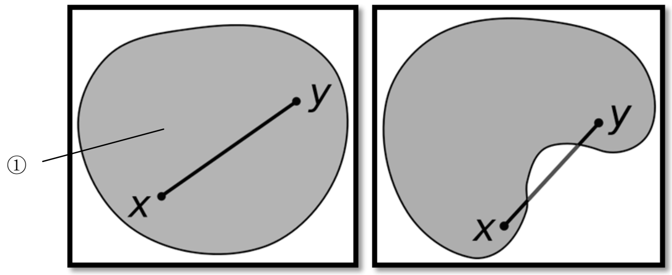


图 4-12　凸集合（左）与凹集合（右）（取自 Wiki 百科）

在图 4-12（左）的集合内部任意两点作直线都一定落在区域①内，所以该区域是凸集合（convex set），而图 4-12（右）则有一些线段会跨出集合区域，所以该区域是非凸集合（non convex set），即凹集合。

在机器学习中，相同类型样本在 n 维特征空间中如果能形成一个凸集合，该样本集就是凸数据集。如图 4-13 所示，左侧是典型的凸数据集，右侧是凹数据集。

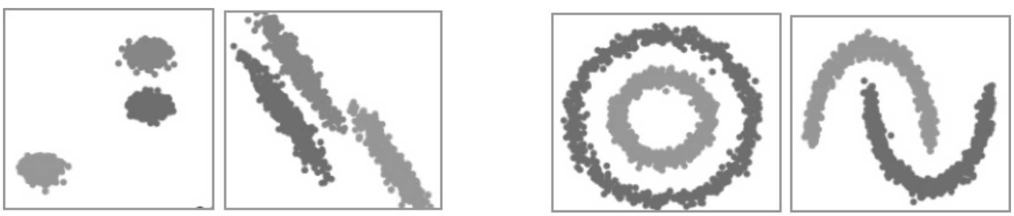


图 4-13　凸数据集（左）与凹数据集（右）

凸/凹数据集的概念不仅用于解决聚类问题，还可以作为选择模型算法的重要依据。

一般来说，凸数据集可以用线性算法解决，比如线性回归、朴素贝叶斯、线性核 SVM 等，采用欧几里得距离的 K-means、AP、GMM 等聚类方法本质上也是线性模型。而对于非凸数据集问题则需要选择非线性模型，比如 RBF 核的 SVM、高斯过程、带核的 K-means、使用非欧距离的 AP 等。密度聚类是一种可以很好地解决非线性聚类问题的聚类模型。

4.5.2 密度算法

密度这个词对每个人来说都不陌生，它是一种衡量单位空间内质量多少的方法。而在机器学习数据集中，密度用来衡量样本在特征空间上分布的紧凑情况。在 DBSCAN 中用下面几种概念表达样本之间的关系。

◎ ε 邻域与邻居：用来衡量两个样本之间的关系。如果两个样本之间的距离小于等于 ε 邻域，则称两个样本户为邻居。

◎ MinPts 与核心点：如果一个样本邻居的数量大于等于 MinPts 个，则称该样本为核心点。在聚类的每个分组中至少有一个核心点，并且不设上限。

◎ 噪声点：如果一个样本的任何邻居和其本身都不是核心点，则称该样本是噪声点。在 DBSCAN 的训练结果中噪声点不属于任何分组。

◎ 边界点：除核心点、噪声点外的所有样本称为边界点。

通过以上概念，DBSCAN 将所有训练样本数据划分成了三种类型：核心点、噪声点、边界点。同时也定义了 DBSCAN 算法的聚类原则：

◎ 将边界点划分到与它互为邻居的核心点相同的分组中。

◎ 如果两个核心点互为邻居，则它们属于一个分组。

◎ 噪声点不属于任何分组。

可以将以上原则想象成一个基于核心点的传递链，互为邻居的核心点越多，则该传递链组成的分组越庞大。而边界点只是附属于核心点，无法继续传递。

因此 DBSCAN 查找这些分组的聚类算法与图论中遍历所有结点的思路非常接近，是一个从给定点寻找核心点并不断扩张当前分组的过程。如果达到无法扩张的状态，则重新选取一个尚未分组的点继续扩张，直到所有样本都被分组。

整个算法在执行过程中有一些细节需要注意，比如如何选取每个分组的第一个样本。一个边界点可能与在不同组内的两个核心点互为邻居，则该边界点只能与它第一个遇到的核心点分在一组。这些因素导致了 DBSCAN 不能像 AffinityPropagation 那样成为一个稳定的算法，在相同数据集上的多次训练结果可能不同。

DBSCAN 的算法虽然很简单，但是在非凸样本集中的表现却非常优秀，如图 4-14 所示。

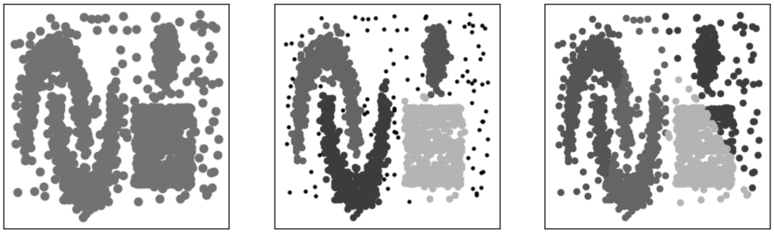


图 4-14　DBSCAN（中）与 K-means（右）在非凸样本集中的表现

如图 4-14 所示，左侧是原始数据集；中图是 DBSCAN 的聚类效果，它用红、蓝、绿、黄很好地划分出了四个比较稠密的整体，黑色小点是识别出的噪声点；而右图是 K-means 在相同数据集上的表现，是明显很差的聚类结果，其原因是凹数据集的存在和噪声样本的影响。

4.5.3　实战：DBSCAN 类

在 sklearn.cluster 包中的 DBSCAN 类封装了 DBSCAN 模型，与其他类的不同主要在于算法相关的初始化参数，具体如下。

- ◎ eps：即 ε 邻域值，该值越大越会倾向于划分出更多的分组。
- ◎ min_samples：即 MinPts 值，可以看成一个分组包含的最少结点数。
- ◎ metric：计算两个样本特征向量距离的方法，默认使用欧几里得距离，但也可以指定 cosine、mahalanobis、minkowski 等其他距离。如 4.1 节所说，如何衡量距离是聚类算法中的一个重要主题，详细内容将在 4.7 节介绍。
- ◎ algorithm：搜索邻居的算法。查找邻居是密度算法的核心步骤，最朴素的方法是对每一个结点轮询其他结点，时间复杂度是 $O(n^2)$。而更好的方法是在结点之间先建立树形结构以加快后续查找，这样总的时间复杂度可以降到 $O(n \cdot \lg n)$。这样的可选算法包括 ball_tree、kd_tree、brute。

◎ leaf_size：当 algorithm 参数选择 ball_tree 或 kd_tree 时的叶子结点大小。

DBSCAN 类在训练之后可以提供数据集的两种信息，一个是常规的 labels_属性，即所有训练样本的分组标签；另一个是所有核心结点的列表，即 components_属性。举例如下：

```
>>>from sklearn import datasets
>>>from sklearn.cluster import DBSCAN
>>>import numpy as np

# 制作训练数据集，由两个随机球型集合组成
>>>blobs1 = datasets.make_blobs(n_samples=10, n_features=2,
                           centers=[[0, 0]] ,cluster_std=[[0.1]])
>>>blobs2 = datasets.make_blobs(n_samples=10, n_features=2,
                           centers=[[1, 1]] ,cluster_std=[[0.1]])
>>>X = np.concatenate((blobs1[0], blobs2[0]))

>>>dbscan = DBSCAN(eps=0.2, min_samples=5).fit(X)      # 初始化模型并训练
>>>dbscan.labels_                                       # 查看聚类结果
[ 0  0  0  0  0  0  -1  0  0  0  1  1  1  1  1  1  -1  1  1  1]

>>>dbscan.components_                                   # 查看核心点
[[ 0.08595481 -0.0230789 ]
 [-0.0065661  -0.02086362]
 [ 0.07948276  0.09764211]
 [-0.03783586 -0.07916153]
 [ 0.00912047  0.10912827]
 [ 1.08595481  0.9769211 ]
 [ 0.9934339   0.97913638]
 [ 1.07948276  1.09764211]
 [ 0.96216414  0.92083847]
 [ 1.00912047  1.10912827]]
```

本例中的数据集由 20 个二维特征样本构成，用参数 eps=0.2、min_samples=5 聚类训练 DBSCAN 后得到标签为 0、1 的两个分组（元素−1 代表噪声点）。通过 components_属性获知共有 10 个核心点，即每个分组包含了不止一个核心点。

4.6 BIRCH

BIRCH 的全称是 Balanced Iterative Reducing and Clustering using Hierarchies，是一种

典型的层次型聚类算法。层次聚类的好处是可以给出聚类后分组之间的亲缘关系，此外由于 BIRCH 在训练的过程中用树形维护在线学习结果，所以它是本章介绍算法中训练时间复杂度最低的模型。BIRCH 的缺点是，只适合凸数据集。

4.6.1　层次模型综述

抛开层次型算法在训练过程中使用树形结构所获得的算法性能提升，众多的层次聚类（BIRCH、Rock、Chameleon）在聚类结果上能提供比之前学习的算法更丰富的内容，即分组之间的关系。

1. 层次知识的意义

在很多场景中，刻画出聚类分组之间的亲缘关系非常有意义。比如一个图书馆想对其所有书籍进行聚类，以便在书架上将门类相近的书籍放在一起供读者参阅。此时除了将金庸的著作“飞雪连天射白鹿，笑书神侠倚碧鸳”分在一起，最好把古龙、梁羽生的书籍也放在附近。而由于都属于文学书籍，这些武侠小说明显应该与《哈利波特》摆放得比《大学英语习题》更近。这些组与组之间距离的知识都可以用树模型表达，如图 4-15 所示。

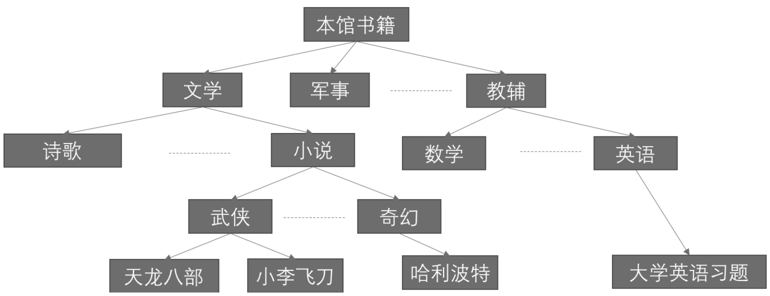


图 4-15　层次模型能表达分组之间的关系

当然，图 4-15 的各中间结点的含义（文学、小说、英语等）无法由模型算法直接给出，层次聚类只能给出类似的层级关系，但这足够所有叶子结点衡量自身与其他所有叶子结点的亲近程度——公共祖先的层次越深则相互的关系越近。

2. 算法分类

虽然层次算法有基本相同的树形分组功能，但出于建树方法的不同和对性能、适配数

据优化的不同，衍生了很多被活跃应用的算法。

从建树策略来看，层次模型可分为如下两大类。

◎ 由底向上策略：最开始将每个样本看成单独的分组，然后逐渐合并。常见的这类算法包括 CURE、BIRCH、ROCK、Chameleon 等。

◎ 由顶向下策略：最开始将所有样本看成一个分组，然后逐渐划分。这类算法的代表是 Bisecting K-means。

在常见的由底向上算法中，BIRCH 效率最高，时间复杂度是 $O(n)$；ROCK 用结点链接代替 BIRCH 中的欧式距离，适用于高维特征样本，时间复杂度是 $O(n^2)$；Chameleon 能够适配非凸数据，时间复杂度是 $O(n \cdot \lg n)$。本节学习在 scikit-learn 中有实现的 BIRCH 模型。

4.6.2 聚类特征树

BIRCH 通过仅扫描一次样本数据，同时逐步建立聚类特征树（Clustering Feature Tree，简称 CF 树）的方式完成训练过程。

说明：由于所有训练数据只使用一次，所以 BIRCH 也具备增强学习（见第 3 章）的能力。

1. 整体结构

所谓聚类特征树是一个类似 B+树的结构，如图 4-16 所示。

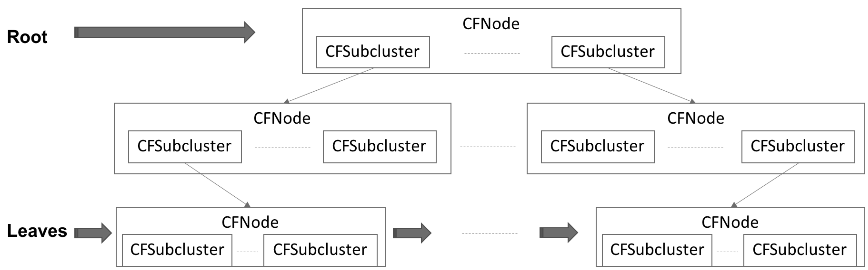


图 4-16　聚类特征树

CF 树的每个结点由若干个 CFSubcluster 组成，每一个 CFSubcluster 就是一个初步的

聚类分组。每个 CFSubcluster 可以有一个子结点，包含该分组内的“子分组”。除常规的根结点指针外，CF 树还用一个链表维护所有叶子结点。树中每个结点内的 CFSubcluster 的最大数量可以由模型参数配置，因此树的高度也会随着训练样本的增加逐渐增长。

2. 聚类特征

BIRCH 将每个聚类特征簇（CFSubcluster）看成有一个中心点的高维球体，因此每个 CFSubcluster 中存储了该球体的特征信息。该特征信息由三个数值完全表达如下。

◎　N 值：该 CFSubcluster 球体中已包含的样本数量。

◎　LS 值：该 CFSubcluster 中所有样本特征向量的线性和，是一个向量。

◎　SS 值；该 CFSubcluster 中所有样本特征向量平方的和，是一个标量。

比如当一个 CFSubcluster 包含了<1, 2>、<2, 3>、<3, 4>这 3 个样本特征向量，则该特征球体的特征值是：N = 3，LS=<1+2+3, 2+3+4> = <6, 9>，SS = <1, 2>•<1, 2> + <2, 3>•<2, 3> + <3, 4>•<3, 4> = 5 + 13 + 25 = 43。

注意：“样本特征向量”与“聚类特征树”两个词中的“特征”含义不一样！

平面上的圆可以由圆心和半径唯一表示，立体空间中的球体也可以通过圆心和半径表示。很容易理解，高维空间中的球体也可以由质心和半径唯一表示，并且它们可以通过 N、LS、SS 值计算获得：

◎　质心 $c = \mathrm{LS}/N$。

◎　半径 $r = \mathrm{SS}/N - 2\cdot C\cdot C + C\cdot C$。

注意：为什么不直接保存质心 c 和半径 r 呢？这是因为 N、LS、SS 有线性可累加性，即当新样本加入时只要把新样本的值加入其中即可；而直接计算 c 和 r 需要用到该簇中之前训练过的所有样本。

3. BIRCH 聚类算法

假设已经有了 CFSubcluster 聚类特征，则通过判定一个样本到该簇质心 c 的距离是否小于半径 r 就可以判断一个样本是否属于某个 CFSubcluste 了。接下来待解决的问题就是 BIRCH 是如何建立特征树并聚类的，它有四个步骤：

◎　用 B+树插入算法逐步建立如图 4-16 所示的特征树。

◎ 将 N 特别小的 CFSubcluster 看成噪声样本，将其从特征树中去除。

◎ 利用其他聚类算法对剩余的 CFSubcluster 聚类，得到更小的特征树。

◎ 用所有 CFSubcluster 的质心 c 作为初始点，按样本距离再次聚类。

上述步骤中 2～4 均为可选步骤，如果不执行这些步骤，那么特征树的每个 CFSubcluste 就是一个聚类分组。

4.6.3 实战：BIRCH 相关调用

scikit-learn 在 sklearn.cluster 中实现了 BIRCH 类，本节解析其超参数和如何读取层次模型。

1. 初始化参数

BIRCH 类的算法相关超参数如下。

◎ threshold：特征树的 CFSubcluster 中高维球体最大半径。新加入样本导致高维球体半径大于该值时，将会导致分裂出新的 CFSubcluster。

◎ branching_factor：每个 CFSubcluster 包含的最大样本数，超过该值导致簇分裂。

◎ n_clusters：聚类分组数量，本参数用于 BIRCH 算法的第四步。如果该值为 None，则直接以每个 CFSubcluster 作为一个聚类分组；如果是整数则定义重新聚类的分组数；也可以是其他聚类类对象，如 K-means、Meanshift 等。

◎ compute_labels：训练后是否提供 labels_属性。如果设为 True，BIRCH 也支持通过 labels_属性读取普通的平铺式聚类分组结果。

2. 层次结果读取

BIRCH 类的训练与预测函数与其他模型类似，独特之处是用树结构表达聚类结果，相关的模型属性如下。

◎ root_：特征树的跟结点，返回一个_CFNode 对象。

◎ dummy_leaf_：特征树第一个叶子结点的指针，返回一个_CFNode 对象。

◎ subcluster_centers_：所有叶子结点中 CFSubcluster 的质心。

◎　subcluster_labels_：叶子结点质心的分组标签。

举例如下：

```
>>>from sklearn.cluster import Birch                    # 引入模型类
# 训练数据
>>>X = [[0, 1], [0.3, 1], [-0.3, 1], [0, -1], [0.3, -1], [-0.3, -1]]

>>>brc = Birch(n_clusters=None).fit(X)                  # 初始化模型并训练

>>>brc.root_                                             # 特征树根结点
<sklearn.cluster.birch._CFNode object at 0x103e67550>

>>>brc.root_.subclusters_                                # 查看根结点有哪些聚类簇
[<sklearn.cluster.birch._CFSubcluster    object    at    0x107794160>,
<sklearn.cluster.birch._CFSubcluster object at 0x10b0bf048>]

>>>brc.labels_                                           # 读取平铺式聚类结果
[0 0 0 1 1 1]
```

如上代码通过初始化一个 n_clusters=None 的 BIRCH 类指定直接将一次样本遍历建立的特征树作为聚类结果。通过 root_属性读到根结点对象，通过根结点的 subclusters_得到两个聚类簇，labels_属性印证了有两个分组的结果。

3. _CFNode 使用

BIRCH 类通过_CFNode 对象描述特征树结点，该对象可以读取的属性如下。

◎　subclusters：该结点中包含的聚类簇 CFSubcluster 列表。

◎　prev_leaf_/ next_leaf_：_下一个/上一个叶子结点，用于通过 dummy_leaf_遍历叶子结点。

◎　centroids_：本_CFNode 所含 CFSubcluster 的质心列表。

◎　sq_norm_：本_CFNode 所含 CFSubcluster 的质心点积列表。

4. _CFSubcluster 使用

_CFNode 的 subclusters 属性返回的是一个_CFSubcluster 对象列表，它保存了真正的聚类信息，可读属性如下。

◎ n_samples/linear_sum_/squared_sum_：高维球的 N、LS、SS 值。

◎ centroid_：本簇质心。

◎ child_：本簇的子结点，是一个_CFNode 对象，叶子结点本值为 None。

这样，递归读取_CFNode->_CFSubcluster->child_->_CFNode->…就可以读取整个层次结构。

思考：聚类算法有哪些？它们有何区别？

4.7 距离计算

聚类的基础是样本与样本之间的距离，大多数聚类算法都有配置特征向量距离计算方式的超参数，比如 AP 算法的 affinity 参数、DBSCAN 的 metric 参数等。它们默认使用欧几里得距离，但有些场景下其他距离衡量方式可能更适合。

4.7.1 闵氏距离

闵氏距离（Minkowski）名称来源于其提出者俄裔德国数学家闵可夫斯基（H.Minkowski），如果您之前没有听说过他，那对他的学生爱因斯坦一定不会陌生。在机器学习中闵氏距离被用来描述等长数值向量之间的距离，设有两个向量 $\langle x_1, x_1, \ldots, x_n \rangle$ 和 $\langle y_1, y_1, \ldots, y_n \rangle$，则闵氏距离计算公式是：

$$D(x,y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}}$$

其中 p 是闵氏距离的次数，随着 p 的不同可以有无数种闵氏距离的计算公式，其中有几个特别的 p 值是最常用的：

◎ $p = 1$ 时，此时的闵氏距离被称为曼哈顿（Manhattan）距离，是所有维度上差值的和，可以将其想象成在繁华都市中只按照正南/正北马路走直线时所经过的距离。

◎ $p = 2$ 时，此时的闵氏距离就是常用的欧几里得（Euclidean）距离，也就是欧几里得在《几何原本》里提出的定理“两点之间直线最短”所指的那条直线。

◎　$p \to \infty$时，闵氏距离被称为切比雪夫（Chebyshev）距离，它的物理意义是将两点之间的距离定义为各维度差值之中的最大值。

如图 4-17 所示是 Wiki 上给出的最直观的 p 处于不同值时的闵氏距离比较。

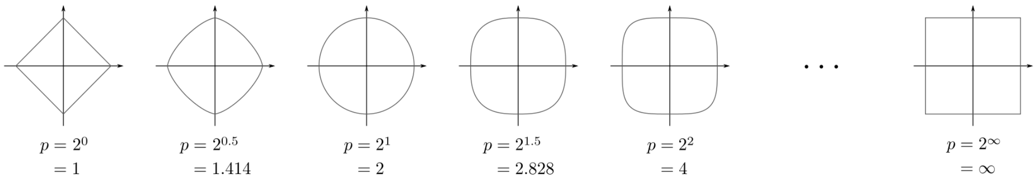


图 4-17　不同 P 值下的闵氏距离

图 4-17 画出的是在二维特征空间中所有距离原点<0, 0>闵氏距离为 1 的点组成的曲线。可以看出，随着 p 值的不断增加，闵氏距离衡量结果只减不增。

注意： 闵氏公式在应用于距离衡量时只取 p>=1；因为当 p<1 时将产生<0, 0>与<1, 1>的距离反而比<0, 0>与<0, 1>小的结果，违背了距离的直观定义。

4.7.2　马氏距离

欧几里得距离与闵氏距离计算方式看似非常符合物理原理，但是在样本特征向量计算中有一些先天的缺陷。

第一，闵氏距离将每个维度的特征看成是等权的，这在有些场景中是致命的缺点。假设有一个只有两个特征（年龄/存款金额）的银行客户聚类系统，有三个样本。

◎　A：<20, 50000>，该客户 20 岁，存款 50000 元。

◎　B：<22, 50500>，该客户 22 岁，存款 50500 元。

◎　C：<60, 50600>，该客户 60 岁，存款 50600 元。

在闵氏距离体系中，计算的结果将会认为客户 A 与 B 的距离大于客户 B 与 C 的距离。但是实际上，年龄 42 岁的差距明显比存款 500 块的差距重要得多，这就涉及不同特征维度应该有不同计算权重的问题。

第二，闵氏距离没有考虑特征之间可能的相关性。还是拿银行客户聚类举例，如果此时将特征“利息”加入系统，是否应该因为利息的不同而增大不同客户之间的距离呢？答案应该是否定的，因为利息的多少实际上取决于存款，此两项特征是高度相关项。而任何

p 值的闵氏距离计算公式都无法指定该相关性。

印度统计学家马哈拉诺比斯提出的马氏距离（Mahalanobis）解决了这两个问题。设有特征向量 $\boldsymbol{x}$、$\boldsymbol{y}$，马氏距离的计算公式是：

$$\boldsymbol{D}(\boldsymbol{x},\boldsymbol{y})=\sqrt{(\boldsymbol{x}-\boldsymbol{y})^{\mathrm{T}}\boldsymbol{S}^{-1}(\boldsymbol{x}-\boldsymbol{y})}$$

其中 $\boldsymbol{S}$ 是特征空间的协方差矩阵。在该协方差矩阵中，对角线上的元素用于表示每个维度的权重，而其他元素表示两两特征维度之间的相关性。

这样，在定义协方差矩阵后，可以通过马氏公式计算出考虑特征权重与相关性的距离结果。

4.7.3　余弦相似度

无论如何调整闵氏距离与马氏距离计算公式中的超参数 p 与 $\boldsymbol{S}$，它们衡量的都是样本在特征空间中的间隔距离。但有的场景中，样本向量之间的间隔并不重要，重要的是不同特征之间的比例关系。这种需要按特征比例关系计算距离的场景多出现在文本分类/聚类中。

文本聚类系统一般是基于单词词频统计的，假设有一个小说书籍聚类任务，该任务样本由武侠、言情两类小说构成，聚类的任务是将这两种小说书籍各自归类。这个任务可以将每本书籍用三个特征来描述："我"字出现的次数、"爱"字出现的次数、"刀"字出现的次数。假设有三本书如下。

◎　A：<54, 298, 5>，即我/爱/刀在全书分别出现了 54/298/5 次。

◎　B：<40, 50, 67>，即我/爱/刀在全书分别出现了 40/50/67 次。

◎　C：<698, 3810, 70>，即我/爱/刀在全书分别出现了 698/3810/70 次。

哪两本书内容上更相似一些呢？通过间隔类距离算法可能得到的结果会是 A 与 B 更近些，因为在数量上它们更相似。但仔细观察就会知道，应该是 A 与 C 更相近一些，因为它们都是"爱字频繁出现、刀字极少出现"的言情小说。造成数值差异的主要原因是 C 的篇幅可能更长一些，但篇幅的长短不应该作为衡量小说种类的依据。

以余弦相似度（cosine similarity）为代表的角度度量标准正是为匹配这种场景而诞生的。设有特征向量 $\langle x_1,x_1,\ldots,x_n\rangle$ 和 $\langle y_1,y_1,\ldots,y_n\rangle$，它们的余弦相似度公式为：

$$S(x,y)=\cos(\theta)=\frac{\sum_{i=1}^{n}(x_i y_i)}{\sqrt{\sum_{i=1}^{n}x_i^2}\cdot\sqrt{\sum_{i=1}^{n}y_i^2}}$$

其中 θ 是两向量之间的夹角，也就是完全用两个向量之间的角度值衡量相似度，角度值越小则相似度越高。

4.7.4　时间序列比较

想象特征向量是对某个变量在一段时间进程中不断采样获得的结果，如何比较这样的两个时间序列向量呢？时间序列向量区别于普通向量的特点是：采样点不一定相同，也就是说特征维度数量可能不同，因此也就意味着两个向量的特征维度不存在一一对应关系，无法使用之前的算法计算。

以动态时间规整（Dynamic Time Wrap，DTW）为代表的时间序列比较算法用于处理这样的问题。它最初被用于处理语音识别问题，因为不同的人语速不同、停顿习惯不同；后来也被广泛应用于金融领域，DTW 很好地处理了交易日并非一一对应、或涨跌时域不同的情况。使用 DTW 的效果如图 4-18 所示。

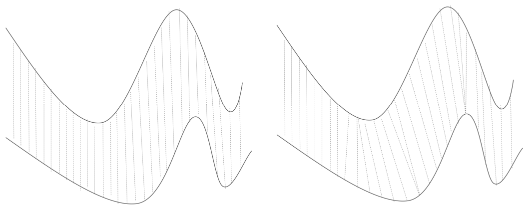


图 4-18　闵式距离（左）与 DTW（右）计算效果

在闵氏和马氏距离中，每个维度有固定的一一对应关系，而 DTW 是在计算了两个向量的所有维度后重新定义每个维度之间的对应关系并计算距离。

DTW 具体算法比较琐碎，超出了本书范围，读者需要时可参考网上的文章，或直接翻阅其提出者于 1978 年发表的论文 *Dynamic programming algorithm optimization for spoken word recognition*。

4.7.5　杰卡德相似度

之前的所有距离都是针对各特征维度在连续数值上的计算。而有时每个特征维度只是

一个布尔量，只有 Yes/No 两种选项，此时可以选择杰卡德相似度（Jaccard similarity）这样的轻量级武器。它的公式非常简单，设有两个向量 $\langle x_1, x_1, \ldots, x_n \rangle$ 和 $\langle y_1, y_1, \ldots, y_n \rangle$，其中向量元素值只能是 Yes/No，则：

$$J(x, y) = \frac{x\text{与}y\text{中都是Yes的维度数量}}{x\text{或}y\text{中是Yes的维度数量}}$$

其含义是两个向量共同特征的比值高低。

假设图书馆想通过读者借过哪些书来对所有读者阅读偏好进行分组。此时可设每个书籍是一个特征维度，因此每个读者都有一个特征向量 $<x_1, x_2, \ldots, x_n>$ 用来表示其借阅过的书籍，其中 n 是图书馆中所有书籍的数量。此时以杰卡德相似度作为距离衡量标准进行任何类型的聚类分组都能获得想要的结果。

4.8 聚类评估

与分类/回归模型用精确度、召回率、均方差等方法评估模型对数据的拟合程度类似，聚类模型也有一些评估算法。但是聚类模型的评估要略显复杂，需要解决两类问题。

一个问题是聚类在训练时没有传入目标值，所以无法用真实值与预测值的直接比较结果作为评估依据。比如某六个样本数据的真实值与预测值是：

```
    true_labels = [1, 1, 2, 2, 0, 0]                    # 真实标签
    pred_lables = [0, 0, 1, 1, 2, 2]                    # 预测标签
```

如果逐一对比样本标签，每一个预测值与真实值都不一样。但这并不意味着这是个失败的聚类预测，在真实标签中有相同值的那些样本在预测标签中也有相同的值。

另一个问题是，如何在没有真实标签的情况下评估模型的优劣。这是大多数聚类应用需要面对的问题，因为如果能拿到真实标签，可能一开始就不会选择聚类这样的无监督学习方法。

1. 带真实值的评估

如果样本数据本身是有真实值标签的，只是在聚类训练时未使用，那么可以用该真实值标签评估聚类效果，但需要处理如前所述的真实标签与预测标签名称不一致的问题。这种情况有几个成熟的指标，scikit-learn 实现了 Adjusted Rand index、Mutual Information、

Homogeneity、completeness、Fowlkes-Mallows 等。它们使用起来非常简单，只需提供真实标签与预测标签序列即可，如：

```
>>>from sklearn import metrics                          # 引入评估包

>>>true_labels = [1, 1, 2, 2, 0, 0]                     # 真实值列表
>>>pred_lables = [0, 0, 1, 1, 2, 2]                     # 测试值列表

# Adjusted Rand index
>>>metrics.adjusted_rand_score(true_labels, pred_lables)
1.0

>>>pred_lables = [1, 1, 2, 2, 1, 0]                     # 另一组测试值列表

>>>metrics.adjusted_rand_score(true_labels, pred_lables)
0.444444444444

# Mutual Information
>>>metrics.adjusted_mutual_info_score(true_labels, pred_lables)
0.465577570605

# Homogeneity
>>>metrics.homogeneity_score(true_labels, pred_lables)
0.710309917857
```

指标 Adjusted Rand index 对适配越好的模型获得的分数越高，最高分是 1，分数低则可能达到负值；其他几种指标也是分数越高越好，具体范围请参考使用手册。

2. 不带真实值的评估

如果样本数据本身没有任何标签，其实是没有任何知识判断聚类结果正确与否的。因此，虽然也有 Silhouette Coefficient、Calinski-Harabaz Index 等进行不带真实值评估的指标，但它们本质上只是从聚类后的分组内部是否紧凑、分组之间界限是否清晰进行形态上的打分，无法真正判断聚类的正确与否。

在使用上，用这类方法进行模型评估需要传入所有样本特征数据和预测值，比如：

```
>>>import numpy as np
>>>from sklearn.cluster import KMeans
>>>from sklearn import metrics

>>>X = np.array([[1, 2], [1, 4], [1, 0],                # 样本特征数据
```

```
            [4, 2], [4, 4], [4, 0]])

>>>kmeans = KMeans(n_clusters=2, random_state=0).fit(X)# 聚类训练
>>>metrics.silhouette_score(X, kmeans.labels_) # Silhouette Coefficient
0.287140797481

# Calinski-Harabaz Index
>>>metrics.calinski_harabaz_score(X, kmeans.labels_)
3.375
```

这类指标的值通常也是越高越好，但注意不要用不同指标的值相互比较。另外，它们通常在 K-means、Affinity Propagation 等产生的凸数据分组上产生较高分数，而对 DBSCAN 等生成的凹数据分组打分较低。

说明：因为有了这些基本评估方法，使得交叉验证工具也可以应用于聚类模型。

4.9 本章内容回顾

◎ 聚类是机器学习的主要应用场景之一，有很多活跃的算法。

◎ K-means 是一种基于划分的聚类算法，在使用时需要指定分组数量，它迭代地找到每个分组合适的中心点。

◎ 近邻算法不需要指定分组数量，且算法具有稳定性，但时间复杂度比 K-means 略高。

◎ 高斯混合模型假设样本整体来自对若干多元高斯分布的采样，因此每个多元高斯分布成为一个聚类分组。

◎ 密度类聚类算法可以很好地适应非凸数据。

◎ BIRCH 是适用于超大数量样本集的层次聚类模型，具备增量学习能力。

◎ 欧式距离就是 p=2 时的闵氏距离；马氏距离考虑了向量维度之间的相关性。

◎ 余弦相似度考虑的是维度之间的比例关系，常用于文本处理、分类。

◎ 时间序列具有维度非一一对应的特点，DTW 是一种时间序列向量比较算法。

◎ 聚类评估方法分为带真实数据评估、不带真实数据评估两类，scikit-learn 提供了较好的封装。

5

第 5 章 无监督学习：数据降维

拜优秀科幻小说《三体》所赐，降维已经成为人尽皆知的科幻用语，降维打击是该小说中的终极武器。在机器学习中，数据降维的目的是将较高维度数据转换为较低维度数据进行表达，同时最大程度上保留原有数据间的关系。本章学习映射高维数据到低维空间的几种降维武器，主要内容如下。

- ◎ 主成分分析（PCA）：使转换后各维度上样本方差最大化的线性降维方法。
- ◎ 线性判别分析（LDA）：与 PCA 类似，但对于带标签的数据能获得更好的结果。
- ◎ 流形基础：什么是流形、局部欧几里得空间、测地线距离等。
- ◎ 等距映射（Isomap）：在降维后保持近邻点间结构不变的流形学习方法。
- ◎ 局部线性嵌入（LLE）：假设样本在局部区域内满足线性关系的流行学习方法。
- ◎ 谱聚类：先用 Laplacian Eigenmaps 进行流形降维、再聚类的方法。
- ◎ t-SNE：用概率建模的流形学习方法。

5.1 主成分分析

主成分分析（Principal Components Analysis，PCA）是统计学中具有较长历史的一种数据集简化技术，在机器学习中常将其用在分类/聚类等模型之前以达到降低特征维度、去除特征之间相关性的目的。

5.1.1 寻找方差最大维度

很多时候在待学习的原始数据中具有大量的冗余特征，对特征数量进行压缩可以提高样本数/特征数比，提高模型的泛化能力。同时，PCA 保证了输出各维度之间的相互独立性，满足了朴素贝叶斯、GMM 等输入特征必须相互独立的模型的要求。

比如在静态图像聚类系统中，每个图像都是一个数据样本，而图像上的每个像素都是一个特征维度。由于图像相邻像素的颜色在多数情况下总是非常接近的，因此在该系统中肯定存在大量的冗余特征。用 PCA 可以将这样的系统的维度数从几千（取决于图像分辨率）降到几百甚至几十，然后在新的维度上进行聚类。

为了便于理解 PCA 的原理，本节分析如何将二维数据降到一维数据，相同的方法可以推广到更高维度的情况。

1. 原始数据

假设现在需要对如图 5-1 所示的二维数据进行 PCA 降维。观察该散点图，请读者先思考这些二维数据是否有冗余信息呢？

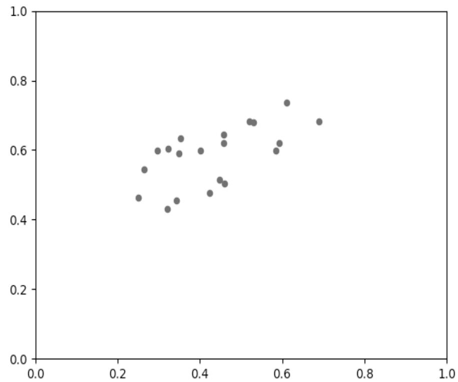


图 5-1 待降维数据

答案应该是“有”，为什么这么说？这是因为可以明显看出该组数据 x 轴与 y 轴是相关的，即 y 轴值大体上随着 x 轴值的增长而增长。这增加了对该组数据进行降维处理的现实意义。

2. 直接删除已有维度实现降维

最直接的降维方式是直接删除一个已有维度，那么要删除横/纵哪一个维度呢？不妨都尝试一下，如图 5-2 所示，分别演示了删除横坐标维度、纵坐标维度的情况。

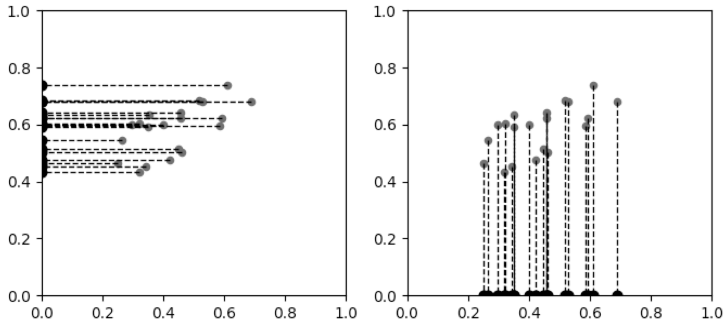


图 5-2　分别删除横/纵（左/右）坐标维度的情况

如果直接删除横坐标，二维平面上的点被退化映射成了 y 轴上的标量，取值范围约为 0.4～0.77，如图 5-2（左）所示；如果删除纵坐标特征，则这些点被映射到了 x 轴上，取值范围约为 0.23~0.7，如图 5-2（右）所示。

思考：应该删除哪个维度呢？

从 PCA 的观点来说，应该选择图 5-2（右）的降维结果，因为该结果在保留的特征维度上实现了更大的样本离散度，即样本方差更大。

3. 寻找方差更大的降维映射

上述降维选择绝不是 PCA 的最终结果，想象如果坐标系原点和方向能够移动，那么在特征空间中就还存在着其他能实现更大样本方差的映射，稍作想象，可以找到的最大方差（最佳）映射轴如图 5-3 所示。

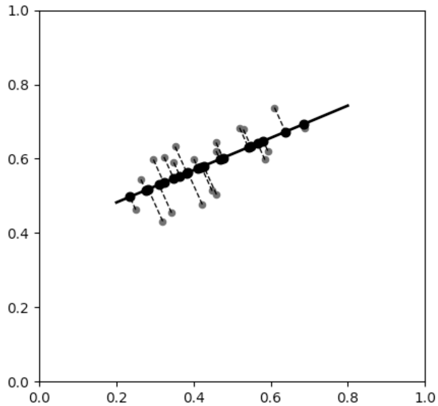


图 5-3　最大方差映射轴

在图 5-3 中的轴上做每个原始点在其上的正交映射就是 PCA 认为能找到的最佳降维方式，因为该映射能获得更加分散的样本数据分布。

4. 用特征值寻找最大方差轴

有很多种方法可以寻找到图 5-3 中的最大方差映射轴，比较常规的是用求样本协方差矩阵较大特征值所对应特征向量的方法，如果想对原始维度为 n 的数据进行 PCA 处理以降维到 m，它的大致流程是：

◎　在各维度上对原始样本进行标准化，即通过线性变换使得样本在各维度上的均值变为零，得到新的样本向量矩阵 $\boldsymbol{X}$ 。

◎　计算原始各维度的协方差矩阵 $\boldsymbol{C}=\frac{1}{n}\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}$，即：如果原始数据包含 $n=30$ 个特征，则生成形状为 30×30 的协方差矩阵。

◎　计算协方差矩阵的特征值和特征向量。

◎　将特征值从大到小排列，取特征值最大的 m 个特征向量组成样本线性映射矩阵 $\tilde{\boldsymbol{U}}^{\mathrm{T}}$ 。

◎　降维后的样本向量矩阵 $\boldsymbol{Y}=\tilde{\boldsymbol{U}}^{\mathrm{T}}\boldsymbol{X}$ 。

5. 用奇异值分解寻找最大方差轴

在线性代数中，矩阵的特征向量其实就是矩阵对应线性变换的一组相互垂直的正交

基，使用协方差矩阵的正交基对原始数据进行降维保证了转换后特征之间的相互独立性。而除了计算协方差矩阵的特征值/特征向量，对原始数据进行奇异值分解（SVD）也能获得一组正交基。可以对任意原始数据矩阵 $\boldsymbol{X}$ 进行奇异值分解：

$$\boldsymbol{X} = \boldsymbol{U\Sigma V}^{\mathrm{T}}$$

其中 $\boldsymbol{U}$ 就是协方差矩阵 $\boldsymbol{C} = \frac{1}{n}\boldsymbol{XX}^{\mathrm{T}}$ 的正交基，而 $\boldsymbol{\Sigma}$ 是奇异值对角矩阵，对角线上的值从大到小排列。这样，无须计算 $\boldsymbol{C} = \frac{1}{n}\boldsymbol{XX}^{\mathrm{T}}$ 本身就可以获得降维映射使用的 m 个样本方差最大的正交基。

说明：关于 PCA 与 SVD 背后的数学原理可参考 https://intoli.com/blog/pca-and-svd/。

5.1.2　用 PCA 降维

在 scikit-learn 的 sklearn.decomposition 包中实现了 PCA 模型，开发者可以直接使用它对数据进行降维处理。其调用方法通过三步完成：

◎　初始化 PCA 对象，同时传入降维目标参数 n_components。

◎　调用 fit()函数用原始数据训练模型。

◎　调用 transform()函数获得降维后的数据。

对 iris 数据集中的数据进行降维处理的代码如下：

```
import matplotlib.pyplot as plt
from sklearn import decomposition                     # 引入包
from sklearn import datasets

iris = datasets.load_iris()                           # 加载 iris 数据库
X = iris.data
y = iris.target

plt.subplot(121)                                      # 绘制降维前的数据
for name, label, m in [('Setosa', 0, "<"), ('Versicolour', 1, "o"), ('Virginica', 2, ">")]:
    plt.scatter(X[y==label, 0], X[y==label,1], label=name, marker=m)
plt.legend()
plt.title("data shape:%s"%(X.shape,))
```

```
    pca = decomposition.PCA(n_components=2)          # 初始化 PCA，降维后维度数是 2
    pca.fit(X)                                       # 训练
    X = pca.transform(X)                             # 获得降维后的数据

    plt.subplot(122)                                 # 绘制降维后的数据
    for name, label, m in [('Setosa', 0, "<"), ('Versicolour', 1, "o"),
('Virginica', 2, ">")]:
        plt.scatter(X[y==label, 0], X[y==label,1], label=name, marker=m)
    plt.legend()
    plt.title("data shape:%s"%(X.shape,))

    plt.show()
```

原始 iris 数据库是一个有四个特征维度的数据集，其中包含了三种花卉的共 150 个样本。以上代码用 matplot 绘制两幅子图，如图 5-4 所示，第一幅子图绘制原始数据中的前两个维度数据，第二幅子图绘制降维后的两个维度数据。

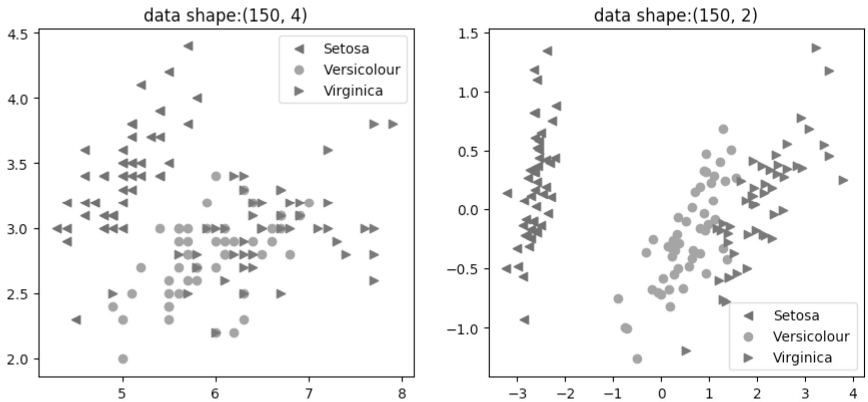


图 5-4　数据集 iris 降维前后

在图 5-4 左图中，由于只是绘制了四个维度中的两个维度，使得 Versicolour 和 Virginica 有大量的样本被迫重叠。在降维后，三类花卉被较好地用二维视图区分开来。

5.1.3　实战：用 PCA 寻找主成分

PCA 类也提供了若干属性用于分析在 PCA 计算过程中产生的正交基向量和它们对应的样本方差。相关属性如下。

◎　components_：样本方差最大的正交基向量。

◎　explained_variance_：协方差矩阵中最大的若干个特征值。

◎　explained_variance_ratio_：若干个最大特征值各自所占的比重。

分析这些属性具有现实意义，比如在人脸特征提取系统中，components_就形成了所谓的“特征脸”，示例代码如下：

```
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_olivetti_faces
from sklearn import decomposition

n_row, n_col = 2, 4
n_components = n_row * n_col
image_shape = (64, 64)

dataset = fetch_olivetti_faces(shuffle=True)   # 在线下载 Olivetti 人脸库
faces = dataset.data
n_samples, n_features = faces.shape
faces -= faces.mean(axis=1).reshape(n_samples, -1)

# 图像绘制函数
def plot_gallery(title, images, n_col=n_col, n_row=n_row):
    plt.figure(figsize=(2. * n_col, 2.26 * n_row))
    plt.suptitle(title, size=16)

    for i, comp in enumerate(images):
        plt.subplot(n_row, n_col, i + 1)
        vmax = max(comp.max(), -comp.min())
        plt.imshow(comp.reshape(image_shape), cmap=plt.cm.gray,
                   interpolation='nearest',
                   vmin=-vmax, vmax=vmax)
        plt.xticks(())
        plt.yticks(())
plt.subplots_adjust(0.01, 0.05, 0.99, 0.93, 0.04, 0.)

plot_gallery("Olivetti faces", faces[:n_components])   # 绘制若干个原始人脸

# 初始化 PCA 模型
estimator = decomposition.PCA(n_components=n_components,
               svd_solver='randomized', whiten=True)
```

```
estimator.fit(faces)                                    # 训练
plot_gallery('Eigen faces', estimator.components_[:n_components])

plt.show()
```

上述代码中的 plot_gallery()函数截取自 scikit-learn 官网代码，在官网的很多例子中用该代码绘制小幅图片。如图 5-5 所示是上述代码绘制的原始人脸。



图 5-5　原始人脸

将图像中的每个像素都作为一个原始特征，可以对整个 Olivetti 进行 PCA 训练。在训练后通过 components_属性可以读取该库图像的若干个“主成分”图像，如图 5-6 所示。

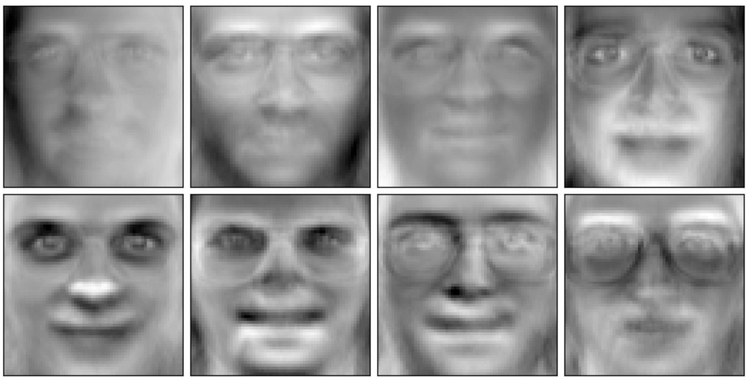


图 5-6　训练后的“主成分”图像

这些 components_不是库里某一个人的脸，但却在最大程度上代表了库中的所有图像。

5.2　线性判别分析

线性判别分析（Linear Discriminant Analysis，LDA）是一种有监督的线性分类器，其原理与 PCA 非常类似，也是将原始特征数据映射到全新的维度，按照各新维度上方差从大到小排序，删除其中较小方差的部分以达到降维的目的。LDA 与 PCA 不同的地方在于其在生成新维度时多考虑了每种标签样本数据集内的分布情况。

注意：机器学习中有两种英文简写为 LDA 的模型，除了本节的 Linear Discriminant Analysis，还有第 8 章将要学习的 Latent Dirichlet Allocation，请读者勿要混淆。

5.2.1　双重标准

因为无监督环境中所有样本数据没有分类标签，所以 PCA 在为样本集定义全新的特征维度时不得不通盘考虑所有样本数据。而 LDA 可以看成在有数据标签时的 PCA 样本，它将寻找新的特征映射正交基时的目标从“最大化所有样本在新维度上的方差”改为了双重标准——“最大化类间样本的方差、最小化类内样本的方差”。

仍然用将二维散点图降维到一维举例，假设发现图 5-1 中的原始数据由两类标签构成，如图 5-7 所示。

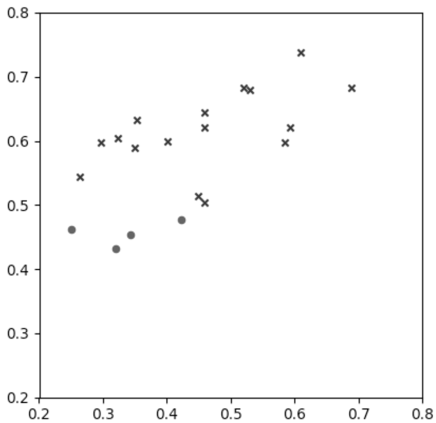


图 5-7　有标签的二维数据

在图 5-7 中，点号和色叉号分别表示每个样本的标签，对该组二维数据分别用 PCA 和 LDA 降维到一维数据，效果如图 5-8 所示。

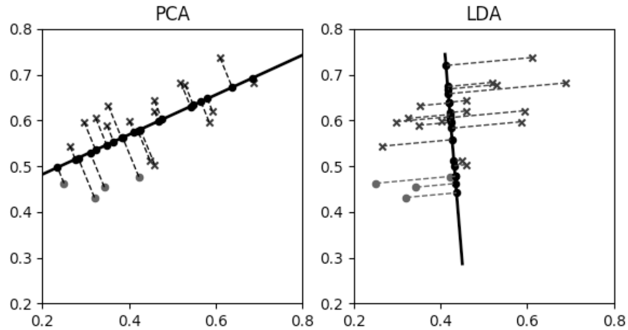


图 5-8　用 PCA 和 LDA 降维到一维数据的效果

图 5-8（左）对该组数据进行 PCA 降维，其效果与图 5-3 保持一致，该映射效果使得新维度上的样本方差最大。图 5-8（右）对原始数据进行 LDA 降维，其效果是使得两类标签数据在新的维度上“最大化类间样本的方差、最小化类内样本的方差”，视觉上看就是使得不同的标签数据尽量不重合！

LDA 是如何达到此效果的呢？其过程与 PCA 也非常类似，设样本数据由 n 种标签的 m 维数据构成，可通过如下几步完成 LDA 计算：

◎ 对每一类标签数据计算其在原始各维度上的均值，得到 n 个 m 维均值向量，记为 $\boldsymbol{U}_1 ... \boldsymbol{U}_n$。

◎ 计算类内原始各维度的协方差矩阵 $\boldsymbol{S}_w = \sum_{i=1}^{n} \boldsymbol{S}_i$，其中 $\boldsymbol{S}_i$ 是每种标签样本的协方差矩阵，$\boldsymbol{S}_w$ 是一个形如 $m \times m$ 的矩阵。

◎ 计算类间协方差矩阵 $\boldsymbol{S}_B = \sum_{i=1}^{n} N_i (\boldsymbol{U}_i - \boldsymbol{U})(\boldsymbol{U}_i - \boldsymbol{U})^{\mathrm{T}}$，其中 N_i 是各标签样本总数，$\boldsymbol{U}_i$ 是各标签均值向量，$\boldsymbol{U}$ 是所有样本的均值向量，$\boldsymbol{S}_B$ 是一个 $n \times n$ 的矩阵。

◎ 计算矩阵 $\boldsymbol{S}_W^{-1} \boldsymbol{S}_B$ 的特征值与特征向量。

◎ 将特征值从大到小排列，取特征值最大的 m 个特征向量组成样本线性映射矩阵 $\tilde{\boldsymbol{U}}^{\mathrm{T}}$。

◎ 降维后的样本向量矩阵 $\boldsymbol{Y} = \tilde{\boldsymbol{U}}^{\mathrm{T}} \boldsymbol{X}$。

建议读者用 5.1.1 中 PCA 的计算步骤对比如上 LDA 的计算步骤，其不同点只是在于如上步骤中的第 2～4 步：PCA 计算的是所有数据协方差矩阵的特征值/特征向量，而 LDA 计算的是类内协方差矩阵的逆与类间协方差矩阵乘积的特征值/特征向量。

此外，由于 $\boldsymbol{S}_w$ 是 M 阶方阵，$\boldsymbol{S}_B$ 是 N 阶方阵，因此用 $\boldsymbol{S}_W^{-1}\boldsymbol{S}_B$ 进行降维后的最大维数既受 N 的约束，也受 M 的约束；而 PCA 降维后的最大维数只受 M 影响。

5.2.2　实战：使用 LinearDiscriminantAnalysis

scikit-Learn 在 sklearn.discriminant_analysis 中提供了 LDA 的封装类 LinearDiscriminantAnalysis，它既是一个有监督学习的分类器，也是一个降维器。作为分类器，它的使用方法与第 3 章中的其他模型一样，通过 fit()、predict()、predict_proba()等函数进行训练和预测。作为一个降维器，其与 PCA 类的使用方式类似，通过 fit()和 transform()函数进行训练和维度转换，只是在调用 fit()时需要多传入样本标签参数。

降维相关的模型初始化参数如下。

◎ Solver：计算新正交基坐标系的方法，可选 svd、lsqr 或 eigen，即奇异值分解、最小二乘法、特征值分解。

◎ n_components：降维后的维度数，必须小于“标签总数−1”。

1. LDA 降维实例

下面是一段对比 PCA 与 LDA 在相同数据集 iris 上进行降维的代码：

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA

iris = datasets.load_iris()                              # 加载数据集
X = iris.data
y = iris.target

plt.subplot(121)
pca = PCA(n_components=2).fit(X)                         # 初始化 PCA 并训练
X_pca = pca.transform(X)                                 # 用 PCA 降维

# 用不同的图例、形状、颜色显示 PCA 降维后的样本
for name, label, m in [('Setosa', 0, "<"), ('Versicolour', 1, "o"),
        ('Virginica', 2, ">")]:
    plt.scatter(X_pca[y==label, 0], X_pca[y==label,1],
```

```
            label=name, marker=m)
plt.legend()
plt.title("PCA")

plt.subplot(122)
lda = LDA( n_components=2).fit(X, y)             # 初始化 LDA 并训练
X_lda = lda.transform(X)                         # 用 LDA 降维

# 用不同的图例、形状、颜色显示 LDA 降维后的样本
for name, label, m in [('Setosa', 0, "<"), ('Versicolour', 1, "o"),
        ('Virginica', 2, ">")]:
    plt.scatter(X_lda[y==label, 0], X_lda[y==label,1],
            label=name, marker=m)
plt.legend()
plt.title("LDA")
plt.show()
```

如上代码的执行效果如图 5-9 所示，其中左图是 PCA 结果，右图是 LDA 结果。

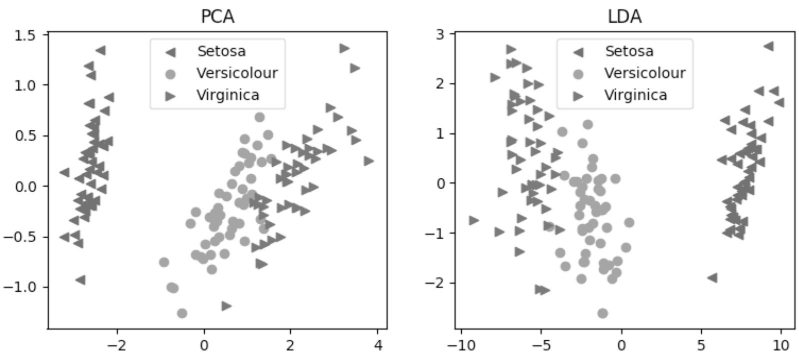


图 5-9　PCA 与 LDA 在 iris 数据集上的降维结果

与 PCA 相比，LDA 在降维后的第一维度（*x* 轴）上保留了更大的标签区分能力，几乎仅凭 *x* 轴的值就可以进行样本分类了（−10～−4 是 Virginica，−4～3 是 Versicolour，3～10 是 Setosa）。

2. explained_variance_ratio_

PCA 和 LDA 模型训练后都有一个属性 explained_variance_ratio_，它用来表示降维后各维度上的方差比。方差比是一个 0～1 的数值，所有维度的方差比和为 1，方差比越高，

说明该维度的信息保留度越高。

LDA 的方差比来源于对 $\boldsymbol{S}_W^{-1}\boldsymbol{S}_B$ 矩阵提取特征值后，各特征值所占特征值总和的比例（奇异值分解同理）。可以读取之前的 iris 数据在降维训练后的方差比：

```
>>>pca = PCA(n_components=2).fit(X)
>>>pca. explained_variance_ratio_
[ 0.92461621  0.05301557]

>>>lda = LDA( n_components=2).fit(X, y)
>>>lda. explained_variance_ratio_
[ 0.99147248  0.00852752]
```

其中 PCA 降维后两个维度的“方差比和”小于 1，说明降维没有完全保留原始信息，有一小部分信息被丢失了；LDA 降维后的“方差比和”等于 1，说明其完全保留了原始信息。

在高维数据无法通过视觉感知降维效果的情况下，方差比是最重要的一种衡量降维效果的指标。通常，可以用“降维保留了百分之多少的方差比”来形容降维效果。

同时，方差比也可以用来指导降维后需要保留的维度数。比如，在一个原始维度为 300 的数据集中，如果发现进行 n_components=10 的降维就可以保留 95%的方差比，而 n_components=30 时只能将该值增加到 96%，此时可以认为 n_components=10 是一个非常合理的选择。

5.3　多维标度法

多维标度法（Multi-Dimensional Scaling，MDS）最初是心理学的一个度量工具，用以理解人们判断的相似性，被广泛应用于市场调研、心理学数据分析。MDS 主要用于解决“已知样本两两之间的距离，如何恢复所有样本坐标”的问题。由于样本之间的距离可来源于任意高维空间，而恢复出的坐标却可以表现在相对低维的空间里，因此 MDS 成为了一种重要的降维模型。

5.3.1　保留距离信息的线性变换

从降维的角度考虑，MDS 与 PCA 有相同的使用场景，它们都是无监督的线性降维方

法。但是它们在线性变换中的优化目标不同，PCA 目标是在新维度上产生最大的样本方差，而 MDS 的目标是“降维前后样本之间的距离尽量保持一致”。

以 MDS 的目标出发，MDS 的降维计算过程如图 5-10 所示。



图 5-10　MDS 的降维计算过程

1. 生成样本距离矩阵 *D*

无论原始数据在怎样高的维度空间中，总能通过计算两两样本之间的距离将其转换成 N 阶样本距离方阵 $\boldsymbol{D}$，其中 N 是样本数量，$\boldsymbol{D}$ 中的元素 d_{ij} 是第 i 个样本与第 j 个样本之间的距离，值越高说明两个样本差异越大。最普通的 d_{ij} 可以采用欧式距离，当然也可以是马氏距离、杰卡德相似度、DTW 等任意其他计量方式。

2. 理解相似度矩阵 *B*

相似度矩阵也是一个 N 阶方阵，其元素 b_{ij} 值越高说明样本 i 和 j 越相似。相似度矩阵中的元素可以通过固定公式 $d_{ij}=-\frac{1}{2}\left(d_{ij}^2-d_{i.}^2-d_{.j}^2+d_{..}^2\right)$ 计算获得，其中 $d_{i.}$ 是 $\boldsymbol{D}$ 中第 i 行的均值，$d_{.j}$ 是 $\boldsymbol{D}$ 中第 j 列的均值，$d_{..}$ 是 $\boldsymbol{D}$ 中所有元素的均值。

此时读者可能会想——为什么要生成相似度矩阵呢？这是因为在向量空间中，内积恰巧就是表示单元向量相似度的方式。也就是说，矩阵 $\boldsymbol{B}$ 就是样本向量的内积矩阵，这样将 $\boldsymbol{B}$ 分解为 $\boldsymbol{XX}^{\mathrm{T}}$ 后，就能得到样本向量 $\boldsymbol{X}$ 了。因此不难理解为什么后续对 $\boldsymbol{B}$ 进行特征值分解后再分解为 $\boldsymbol{XX}^{\mathrm{T}}$ 形式可以达到降维的目的。

技巧：向量内积可用于描述样本之间相似度的性质，在机器学习中非常重要。回忆非线性学习中重要的核函数概念，它也是在高维空间中对样本内积的表达。

3. 特征值分解提取主要成分

在图 5-10 中，第 3～5 步骤的作用与 PCA 和 LDA 中的特征值分解作用一样，都是要提取被分解矩阵的主要成分。通过保留大的特征值对应的特征向量，也就是说，可以获得主成分相似度矩阵 $\boldsymbol{B}^*$，直接对其分解为 $\boldsymbol{XX}^{\mathrm{T}}$ 后即可获得降维后的样本集 $\boldsymbol{X}$。

可以通过特征值分解完成重要的 $\boldsymbol{B}^* \to \boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}$ 计算，其基本步骤是：

◎　将 $\boldsymbol{B}^*$ 特征值分解为 $\boldsymbol{B}^* = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1}$ 的形式，其中 $\boldsymbol{Q}$ 是特征向量，$\boldsymbol{\Lambda}$ 是特征值对角矩阵 $\begin{bmatrix} \lambda_1 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & \lambda_{\mathrm{m}} \end{bmatrix}$，$\lambda_1 \ldots \lambda_m$ 是特征值，m 是降维后的维度数。

◎　因此 $\boldsymbol{B}^* = \boldsymbol{Q}\begin{bmatrix} \sqrt{\lambda_1} & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & \sqrt{\lambda_{\mathrm{m}}} \end{bmatrix}\begin{bmatrix} \sqrt{\lambda_1} & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & \sqrt{\lambda_{\mathrm{m}}} \end{bmatrix}\boldsymbol{Q}^{-1}$。

◎　所以 $\boldsymbol{X} = \boldsymbol{Q}\begin{bmatrix} \sqrt{\lambda_1} & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & \sqrt{\lambda_{\mathrm{m}}} \end{bmatrix}$。

5.3.2　MDS 的重要变形

从严格意义上说，在上一节中描述的 MDS 流程是传统的 metric MDS，自然还有和它对应的 non-metric MDS。此外，本节还讨论 MDS 与 PCA 的内在联系。

1. 非计量 MDS（non-metric MDS）

传统 metric MDS 精确地计量样本之间的距离并在降维中保留该距离。但是有些场景中只关心样本间距离的大小顺序，而不是精确的距离值，这样的 MDS 被称为 non-metric MDS，简称为 NMDS；传统 metric MDS 有时也被简称为 CMDS（classic MDS）。

比如，假设原始特征空间中的三个样本 a、b、c 之间有距离关系 distance(a, b) = distance(a, c) + p，其中 p 是任意正值，则在降维后只需保证距离不等式 distance(a, b) > distance(a, c)，而无须确切地保留 p 值。

对于这样的降维需求，在 NMDS 中定义了损失函数 STRESS 作为优化目标控制距离矩阵 $\boldsymbol{D}$ 的计算。STRESS 基于样本之间的距离迭代计算而来，NMDS 的目标是使 STRESS 最小化，有兴趣的读者可参阅 Michael A. A. Cox 的论文 *Interpretation of Stress in non-metric Multidimensional Scaling*。

一个比较普及的最小化 STRESS 的算法称为 SMACOF（Stress Majorization of a Complicated Function）。由于 metric MDS 可以看成 NMDS 的极端情况，STRESS 和 SMACOF

后来也被用来计算 metric MDS。

2. CMDS 与 PCA 的关系

MDS 与 PCA 都是无监督的降维模型，且在计算过程中都用到了特征值分解后提取主要特征向量，它们之间是否有什么直接联系呢？答案是肯定的。

在 MDS 中被特征值分解的矩阵是样本内积矩阵 ***B***，而在 PCA 中被特征值分解的是均值化后样本的协方差矩阵 ***C***。如果在 MDS 计算距离矩阵 ***D*** 的时候使用欧式距离，则内积矩阵 ***B*** 可以通过线性变换转换成协方差矩阵 ***C***。

因此，可以将 PCA 看成 CMDS 在使用欧式公式计算原始样本距离时的一种特殊情况，PCA 与使用欧式距离的 CMDS 在降维效果上没有区别。

5.3.3 实战：使用 MDS 类

在 scikit-learn 的 sklearn.manifold 中实现了 MDS 模型，通过调用 fit_transform()函数可完成降维转换，该模型比较重要的初始化参数如下。

- ◎ n_components：降维后的维度数。
- ◎ metric：True 或 False，表示使用 metric MDS 算法或 non-metric MDS 算法。
- ◎ n_init、max_iter、eps：在 SMACOF 算法中控制 STRESS 计算的参数。
- ◎ dissimilarity：euclidean 或 precomputed，如果是 euclidean 则在执行 fit()函数时传入原始特征向量；如果是 precomputed 则在执行 fit()函数时传入计算好的距离矩阵 ***D***，当然此时可以用任何距离公式计算该矩阵。

如下示例代码演示 MDS 的调用方式，并对比 MDS 与 PCA 在相同数据集上的效果：

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.manifold import MDS                          # 引入 MDS 包

iris = datasets.load_iris()
X = iris.data
y = iris.target

####### 此处省略 PCA 降维及绘制代码，请参考本章第一节 #########
```

```
plt.subplot(122)
mds = MDS( n_components=2, metric=True)            # 初始化metric MDS对象
X_mds = mds.fit_transform(X)                       # 降维转换

for name, label, m in [('Setosa', 0, "<"), ('Versicolour', 1, "o"),
      ('Virginica', 2, ">")]:
   plt.scatter(X_mds[y==label, 0], X_mds[y==label,1],
         label=name, marker=m)
plt.legend()
plt.title("MDS")
plt.show()
```

仍然用 iris 数据集比较两个模型的降维效果，上述代码的执行效果如图 5-11 所示，其中左图是 PCA，右图是 MDS。虽然乍看上去结果不太一样，但是请读者想象将左图左右翻转后稍作逆时针旋转的效果，其结果就与右图非常相近了！

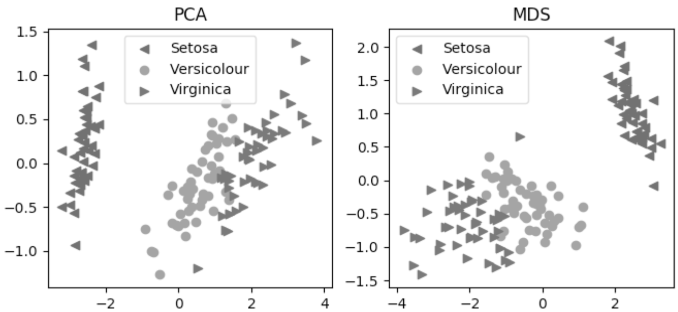


图 5-11　PCA 与 MDS 降维对比

在 5.3.2 节中分析了 metric MDS 与 PCA 本质的一致性，为什么实践中会产生这样需要线性变换才能匹配的差别呢？这是因为 MDS 将原始坐标转换成距离矩阵后进行降维，而这一转换丢失了原始数据中各样本的方位特性。表现在结果上即降维后的数据方向无法与 PCA 保持一致。

思考：MDS 有哪些变形？

5.4　流形学习之 Isomap

流形学习是非线性降维的主要方法，从本节开始讲解流形学习的基本原理与实践。Isomap 是 MDS 在流形语境下的直接扩展，是最好的流形入门模型。

5.4.1 什么是流形

按照 Wiki 上的定义，流形（manifold）是局部具有欧几里得性质的空间，通常嵌入比该局部空间更高维的外围非欧几里得空间中。如果觉得该定义比较晦涩，那么分析流形一词的英文含义就显得非常直白了：

$$\text{Manifold(流形)} = \text{many} + \text{fold(小平面)}$$

也就是说流形是若干个欧几里得空间的拼接，这里的“Many”可以是有限的个数，也可以是无限多个。

1. 非欧几里得空间

流形概念本身来自于用数学语言描述非欧几里得几何空间的需求，其最著名的应用莫过于爱因斯坦用“三维空间+时间”的四维流形解释广义相对论了。那么什么是非欧几里得空间呢？

大部分人对世界几何空间的认知是构建在欧几里得理论上的，从来不会怀疑“两点之间线段最短”“两条平行线永不相交”“三角形内角之和是 180 度”等基本定理的正确性。

但有些时候这些似乎恒久不变的真理却异常脆弱，从宏观角度观察人类生活的地球，就会发现它其实不是一个欧几里得空间。想象地球上任意的两条经度线，它们在赤道附近都是平行线，但是所有经度线其实最终都相交于南北两极；从宏观来看，在地球上相距较远的三个顶点作三角形（比如一个角在北极、另两个角在赤道），其内角和也不是 180 度。

这样就可以理解，整个地球本身是一个非欧几里得空间，但这不影响用欧几里得方法分析地球上的局部地区。因此可以把地球表面看成一个嵌入在三维外围模型中的二维流形。

2. 流形学习

流形学习的概念于 2000 年在 *Science* 杂志上被提出，其代表算法是 Isomap 和 LLE。流形学习的目的就是将嵌入在外围空间（Ambient Space）中的流形提取到与其内蕴空间（Intrinsic Space）同构的新的低维空间中。

仍用地球作为一个流形举例：以宇宙视角俯视的三维空间是地球所在的外围空间，如

果忽略地球不同地区的海拔差异，则地球表面的局部区域就是二维的内蕴空间，而二维地图则可看成对地球进行流形学习后的降维结果。

当然，对于地球仪这样理想的简单流形，仅通过简单的坐标映射就可以完成从三维空间到二维空间的转换，根本不需要动用流形学习这样的重型武器。流形学习真正要解决的是各种嵌入在高维空间中的不规则流形，如图 5-12 所示。

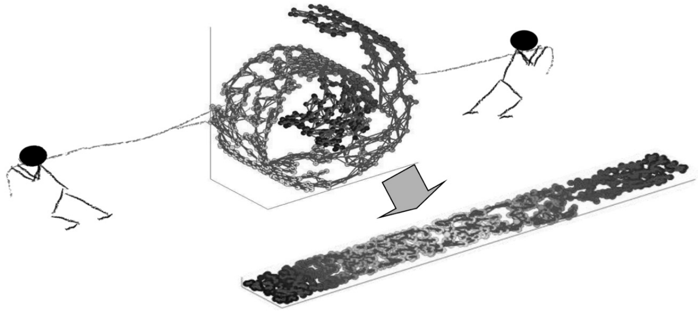


图 5-12　流形学习示意图（该图取自互联网）

在图 5-12 中的流形是一条卷曲在三维空间中的二维彩带，流形学习的理想结果是将其展开平铺在二维空间中。这样的流形学习一般可以通过两步来完成：

◎　识别出外围空间中组成局部欧几里得空间的样本，并计算样本之间的距离。

◎　将这些距离等距重构（isometry）到低维空间完成降维，MDS 就是典型的等距重构算法。

3. 适用性分析

了解了流形学习的意图及原理后，可以初步分析流形降维的适用性了。

首先，就像 PCA 等要求高维空间本身有信息冗余，流形学习也仅适用于流形嵌入比较明显的场景。

其次，流形的基础是局部的欧几里得空间，所以在流形学习中要求有足够密集的数据样本能让算法检测到局部空间。就像图 5-12 中的样本数据几乎是一整条彩带而没有隔断。

最后，流形学习一般对噪声比较敏感，如果在图 5-12 中的彩带卷中间有噪声数据，则可能会被识别为堆状流形，而非带状流形。

5.4.2 测地线距离

Isomap 是 Joshua B. Tenenbaum 于 2000 年在 *Science* 上发表的流形学习模型，它的基本原理是用测地线计算样本之间的欧几里得距离，然后用该距离进行 MDS 降维。读者应该已经对 MDS 相当熟悉，本节讨论测地线的概念与计算原理。

测地线距离（Geodesic Distance）是沿着流形内蕴空间测量的距离。如图 5-13 所示是 Joshua 用来解释 Isomap 的经典图例，其中左图的实线示意了理想测地线，而与其相对的虚线示意的是外围空间中的“最短路径”。这里的测地线距离相当于在地球上衡量从中国上海到美国洛杉矶之间沿着太平洋航行的距离，而不是挖超级隧道能连接两个城市的最短距离，这也是测地线一词的由来。

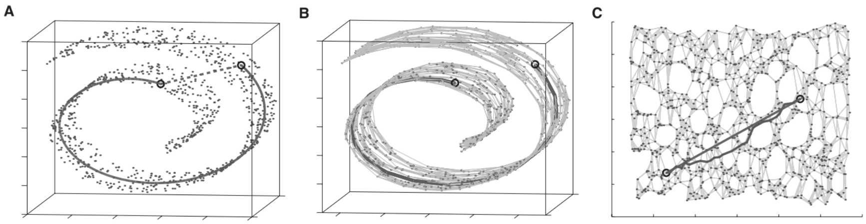


图 5-13 测地线示意图（取自 Joshua 的原始 Isomap 论文）

而图 5-13 的中图实线示意的是 Isomap 能通过训练样本计算出的近似测地线。由于数据样本的有限性，在流形学习中是没办法像在平地上走路一样找出绝对理想的测地线的，而只能妥协地沿着样本找到近似的测地线。这有点像在池塘中沿着荷叶跳跃的青蛙，如图 5-14 所示，其中的荷叶就是已知的训练样本数据。

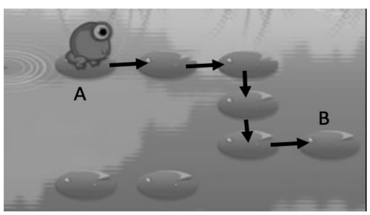


图 5-14 用近似测地线衡量 A、B 两点距离

图 5-13 的右图则是在流形展开后对理想测地线与近似测地线的比较。样本越密集，近似测地线就能越好地拟合理想测地线。

理解了 Isomap 的近似测地线后，Isomap 流形降维的三步流程一目了然。

◎ 找出所有样本点的邻近样本点，它定义了流形理论中的“局部欧几里得空间”。邻近点可以用“最近的 k 个邻居”或“距离 d 以内的所有邻居”等策略选取。

◎ 利用上一步建立的临近点图，计算所有样本点之间的最短路径，生成测地线距离矩阵。最短路径可以用 Dijkstra、Floyd 等图论经典算法计算。

◎ 将测地线距离矩阵输入 MDS/kernelPCA 模型，完成降维。

KernelPCA 是用核函数计算样本协方差矩阵的 PCA 算法，使 PCA 也有了非线性学习能力。之前讨论过 CMDS 与 PCA 的本质一致性；相应地，非线性的 MDS 与 kernelPCA 也有本质一致性。因此在实践中，MDS 和 kernelPCA 都可以作为 Isomap 的降维算法。

5.4.3　实战：使用 Isomap 类

在 scikit-learn 的 sklearn.manifold 中实现了 Isomap 模型，通过 fit()、transform()等函数可以直接完成 Isomap 的训练和降维。与算法有关的模型初始化参数如下。

◎ n_neighbors：寻找样本近邻时“最近的 k 个邻居”中 k 的值。

◎ n_components：降维后的维度数量。

◎ eigen_solver：计算矩阵特征值的算法，arpack 使用 Arnoldi 迭代算法，适用于稀疏矩阵，dense 使用直接解法，适用于稠密矩阵。

◎ tol、max_iter：控制 Arnoldi 算法的迭代参数。

◎ path_method：测地线最短路径的生成算法，FW 使用 Floyd-Warshall，时间复杂度是 $O(N^3)$。$\boldsymbol{D}$ 使用 Dijkstra 算法，时间复杂度是 $O(N^2(k+\log(N)))$，其中 N 是样本数量，k 是原始维度数。

◎ neighbors_algorithm：近邻搜索算法，可选 brute、kd_tree、ball_tree。

在模型训练后，可以通过如下属性获得学习结果。

◎ embedding_：样本降维后的结果。

◎ kernel_pca_：scikit-learn 使用 KernelPCA 作为降维算法，可以通过本属性读取 KernelPCA 对象。

◎ nbrs_：以 sklearn.neighbors.NearestNeighbors 对象表达的样本近邻结构。

◎ dist_matrix_：测地线距离矩阵。

如下代码演示了在设置不同 n_neighbors 的情况下 Isomap 在 iris 数据集上的降维表现：

```
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.manifold import Isomap                         # 加载 Isomap 模型

iris = datasets.load_iris()                                 # 加载 iris 数据库
X, y = iris.data, iris.target

plt.subplot(221)                                            # 训练并绘制 PCA 效果
X_pca = PCA( n_components=2).pca.fit_transform(X)
for name, label, m in [('Setosa', 0, "<"), ('Versicolour', 1, "o"),
      ('Virginica', 2, ">")]:
   plt.scatter(X_pca[y==label, 0], X_pca[y==label,1],
         label=name, marker=m)
plt.legend()
plt.title("PCA")

# 选取三种不同的 n_neighbors
for idx, neighbor in enumerate([2, 20, 100]):
   plt.subplot(222 + idx)
   # 初始化 Isomap 模型
   isomap = Isomap( n_components=2, n_neighbors=neighbor)
   X_isomap = isomap.fit_transform(X)                       # Isomap 降维

   for name, label, m in [('Setosa', 0, "<"), ('Versicolour', 1, "o")
         , ('Virginica', 2, ">")]:
      plt.scatter(X_isomap[y==label, 0], X_isomap[y==label,1],
            label=name, marker=m)
   plt.legend()
   plt.title("Isomap (n_neighbors=%d)"%neighbor)

plt.show()
```

代码效果如图 5-15 所示。n_neighbors 值越小，流形上的局部欧几里得空间越小，降维后样本间的排列效果越明显；n_neighbors 值越大，测地线距离越接近外围空间的欧几里得距离，使得 Isomap 更接近普通 PCA 的结果。

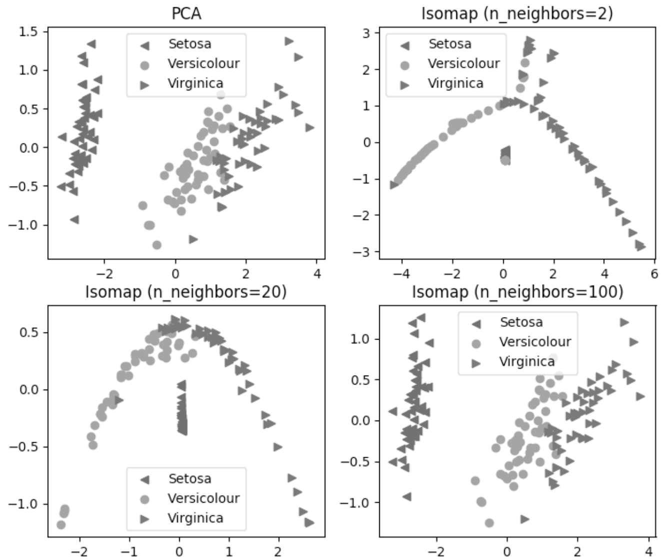


图 5-15　不同 n_neighbors 下 Isomap 在 iris 数据集上的降维表现

当近邻数是 2 时，样本间只能“一个接一个”地看到其他样本，因此降维后呈右上图效果；而当近邻数达到 100 时，Isomap 与 PCA 的效果已经完全一样。

5.5　流形学习之局部嵌入

Isomap 在降维中需要计算所有样本之间的测地线距离。与其相对的，本节的局部嵌入是一簇只考虑相互邻近样本间关系的模型，其中局部线性嵌入（LLE）与 Isomap 在很短的时间段内相继在 2000 年被发表，其后又衍生出拉普拉斯特征映射（LE）等多种其他局部嵌入类算法。

5.5.1　局部线性嵌入

局部线性嵌入（Locally Linear Embedding，LLE）是一种以保持每个样本与邻近样本之间距离作为降维目标的流形学习方法。它通过一个邻近权重系数矩阵 $\boldsymbol{W}$ 表达每个样本与邻近样本之间的关系，并试图在降维后保持该权重矩阵。LLE 算法的提出者 Lawrence K. Saul 形象地用图 5-16 描述了 LLE 原理。

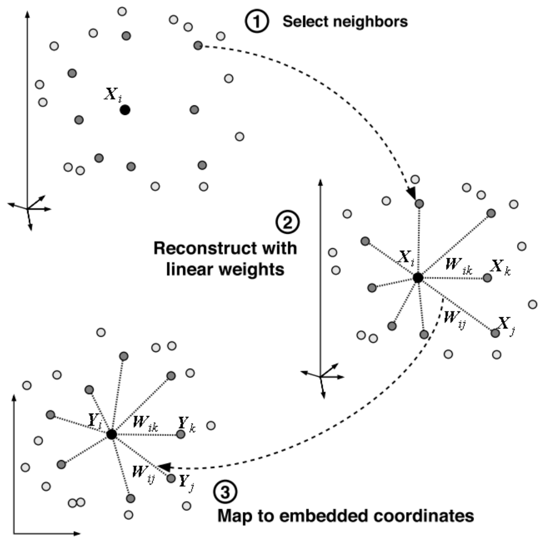


图 5-16　LLE 原理

根据图中描述，LLE 降维过程包括三步：选择近邻→建立权重系数矩阵→降维。其中第一步几乎存在于任何流形学习算法中，下面主要描述后面两步。

1. 权重系数矩阵

LLE 的基本假设是每个样本可以通过邻近结点的线性组合近似地表示。权重系数矩阵中的每个元素值 w_{ij} 用于表示用样本 j 生成样本 i 时使用的权值，即

$$\overline{\boldsymbol{x}_i} = \sum_{j \in N_i} w_{ij} \boldsymbol{x}_j$$

其中 N_i 是样本 i 所有邻近点的集合，可以用“最近的 k 个点”这样的策略定义。$\boldsymbol{x}_j$ 是第 j 个真实的原始特征向量，$\overline{\boldsymbol{x}_i}$ 是第 i 个通过邻近点恢复出的特征向量。

2. 寻找最优权重

需注意上述公式中 $\overline{\boldsymbol{x}_i}$ 与真实的第 i 个样本特征向量 $\boldsymbol{x}_j$ 的区别。理想情况下 $\boldsymbol{x}_i$ 等于 $\overline{\boldsymbol{x}_i}$；但一个样本 i 是否真的能够通过其附近结点的线性组合恢复，这取决于样本自身的分布情况。因此 LLE 首先需要找出能够使 $\overline{\boldsymbol{x}_i}$ 最接近 $\boldsymbol{x}_i$ 的一组权重系数，也就是如下的优化目标。

$$\min\sum_i \left\| \boldsymbol{x}_i - \overline{\boldsymbol{x}_i} \right\|^2$$

注意：双竖线 $\|\boldsymbol{v}\|$ 是度量符号，在欧几里得空间中取模符号 $|\boldsymbol{v}|$ 相当，是向量各分量平方和的开方运算。比如 $\|\langle 2,-1,3\rangle\| = \sqrt{4+1+9} = \sqrt{14}$，其物理意义是向量在欧几里得空间中的长度。

再加上权重归一化要求 $\sum_{j\in N_i} w_{ij} = 1$，该带约束条件的最小化目标可以通过拉格朗日乘子法求解，在第 3 章的 SVM 模型中也曾使用过该方法。

3. 降维

通过前一步，已经有了在高维空间中的权重系数矩阵 $\boldsymbol{W}$，现在的目标是找到低维空间中所有样本的向量值，使得 $\boldsymbol{W}$ 尽量保持不变。用公式表达，也就是最小化优化目标：

$$\mathrm{J}(Y) = \sum_{i=1..n} \left\| y_i - \sum_{j\in N_i} w_{ij} x_j \right\|^2$$

其中 y_i 是低维空间中的第 i 个样本，N_i 是 i 的近邻样本，n 是样本数量。在此优化目标中，w_{ij} 是已知变量，y_i 是未知变量。设 $\boldsymbol{I}$ 是单位矩阵，$\boldsymbol{Y}$ 是降维后的样本矩阵，上式最终可以转换为：

$$J(\boldsymbol{Y}) = \mathrm{trace}(\boldsymbol{Y}^{\mathrm{T}}(\boldsymbol{I}-\boldsymbol{W})^{\mathrm{T}}(\boldsymbol{I}-\boldsymbol{W})\boldsymbol{Y})$$

设 $\boldsymbol{M} = (\boldsymbol{I}-\boldsymbol{W})^{\mathrm{T}}(\boldsymbol{I}-\boldsymbol{W})$，则有 $J(\boldsymbol{Y}) = \mathrm{trace}(\boldsymbol{Y}^{\mathrm{T}}\boldsymbol{M}\boldsymbol{Y})$，另有归一化条件 $\boldsymbol{Y}^{\mathrm{T}}\boldsymbol{Y} = n\boldsymbol{I}$，降维结果 $\boldsymbol{Y}$ 同样可以通过拉格朗日乘子求解。

注意：trace()是矩阵迹函数，即矩阵对角线元素之和。

4. 用特征值分解求优化问题

截至上一步，已经给出了从高维空间 $\boldsymbol{X}$ 样本矩阵到低维空间 $\boldsymbol{Y}$ 矩阵的完整计算步骤。但最后一步通常使用对 $\boldsymbol{M}$ 进行特征值分解替代常规拉格朗日乘子法：

◎ 因为，形如 $\min(\mathrm{trance}(\boldsymbol{Y}^{\mathrm{T}}\boldsymbol{M}\boldsymbol{Y}))$ 或 $\min(\mathrm{trance}(\boldsymbol{Y}^{\mathrm{T}}\boldsymbol{M}\boldsymbol{Y}))$ 这类拉格朗日乘子求解问题最终可以推导出等式 $\boldsymbol{M}\boldsymbol{Y} = \varLambda\boldsymbol{Y}$，其中 $\varLambda$ 是 $\boldsymbol{M}$ 的特征值组合，而 $\boldsymbol{Y}$ 就是 $\boldsymbol{M}$ 的特征向量矩阵。

◎ 所以，只要对 $\boldsymbol{M}$ 进行特征值分解并取最小（当优化目标是 min 时）或最大（当优化目标是 max 时）的 d 个特征值对应的特征向量（d 是降维目标空间中的维度数），就可直接求得 $\boldsymbol{Y}$。

5.5.2 拉普拉斯特征映射（LE）

拉普拉斯特征映射（Laplacian Eigenmaps，LE）是与 LLE 非常像的一个局部嵌入降维方法。它们的不同之处在于 LLE 围绕尽量保持权重系数的矩阵进行降维，而 LE 围绕保持拉普拉斯的矩阵（Laplacian Matrix）进行降维。

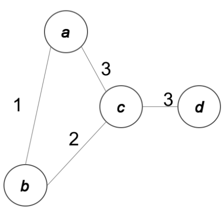
注意：虽然 LLE 与 LE 是同一类流形学习方法，英文简写也接近，但英文全称完全不同！

1. 拉普拉斯矩阵

拉普拉斯矩阵是图论中一种用于表示无向有权图结构的矩阵，它由两部分组成，即拉普拉斯矩阵 $\boldsymbol{L}=\boldsymbol{D}-\boldsymbol{W}$，其中 $\boldsymbol{D}$ 是度矩阵，$\boldsymbol{W}$ 是邻接权重矩阵。$\boldsymbol{W}$ 中的每个元素是两个点之间边的权值；而 $\boldsymbol{D}$ 是对角矩阵，其对角线元素是该点所有边的权值和，即 $d_{ii}=\sum w_{ij}$。

如图 5-17 所示是一个由无向图生成拉普拉斯矩阵的例子，该图由四个顶点和四条边组成，边上的数字是该表边的权重。在分别计算 $\boldsymbol{D}$ 和 $\boldsymbol{W}$ 后，就可用公式得到 $\boldsymbol{L}$。

图



*D*矩阵

	a	b	c	d
a	4	0	0	0
b	0	3	0	0
c	0	0	8	0
d	0	0	0	3

*W*矩阵

	a	b	c	d
a	0	1	3	0
b	1	0	2	0
c	3	2	0	3
d	0	0	3	0

*L*矩阵

	a	b	c	d
a	4	-1	-3	0
b	-1	3	-2	0
c	-3	-2	8	-3
d	0	0	-3	3

图 5-17　拉普拉斯矩阵举例

2. LE 中权值的计算

在矩阵 $\boldsymbol{L}=\boldsymbol{D}-\boldsymbol{W}$ 中，$\boldsymbol{D}$ 可以由 $\boldsymbol{W}$ 计算而来，所以原始样本的拉普拉斯矩阵完全取决于 $\boldsymbol{W}$ 的计算。而 LE 中，$\boldsymbol{W}$ 矩阵中的值用来表示样本之间的相似度，即距离越近的样本点权值越高，具体可以有如下三种策略。

◎　只保留距离最近的 k 个近邻的相似度，其他较远样本权值设为 0。

◎　只保留距离为 ε 以内的近邻的相似度，其他较远样本权值设为 0。

◎　保留每个结点与所有其他结点的相似度。

其中前两种策略都是“局部”策略，而第三种退化为一种类似 Isomap 的全局策略。因为局部策略会导致 $\boldsymbol{W}$ 中存在大量的 0 而成为稀疏矩阵，在后续计算中有很大的性能优势。

3. 降维优化函数

设样本降维后的 d 维特征向量用 $\boldsymbol{y}$ 表示，LE 的目标则是寻找到合适的 $\boldsymbol{y}$，有如下最小化优化目标：

$$J(\boldsymbol{Y})=\sum_{i,j\in\boldsymbol{N}}\left\|\boldsymbol{y}_i-\boldsymbol{y}_j\right\|^2 w_{ij}$$

其中 $\boldsymbol{N}$ 是所有样本集合，$\left\|\boldsymbol{y}_i-\boldsymbol{y}_j\right\|^2$ 可以看成两个样本的距离度量，所以该目标的直观意义可以被描述成：

◎　两样本之间原始的权值 w 越高，则降维后越应该使两个样本距离小，即离得近。

◎　样本间权值越低，则它们降维后的距离对整体目标影响越小，也就是说可以离得更远。

将该目标函数整理后可以发现：

$$J(\boldsymbol{Y})=\sum_{i,j\in\boldsymbol{N}}\left\|\boldsymbol{y}_i-\boldsymbol{y}_j\right\|^2 w_{ij}=\sum_{i,j\in\boldsymbol{N}}\boldsymbol{y}_i^{\,2}w_{ij}+\sum_{i,j\in\boldsymbol{N}}\boldsymbol{y}_j^{\,2}w_{ij}-2\sum_{i,j\in\boldsymbol{N}}2\boldsymbol{y}_i\boldsymbol{y}_j w_{ij}$$

由于拉普拉斯矩阵中 $d_{ii}=\sum w_{ij}$，所以目标优化函数最终变成：

$$\begin{aligned}J(\boldsymbol{Y})&=\sum_{i\in\boldsymbol{N}}\boldsymbol{y}_i^{\,2}d_{ii}+\sum_{j\in\boldsymbol{N}}\boldsymbol{y}_j^{\,2}d_{jj}-2\sum_{i,j\in\boldsymbol{N}}2\boldsymbol{y}_i\boldsymbol{y}_j w_{ij}\\&=\operatorname{trace}(2\boldsymbol{Y}^{\mathrm{T}}(\boldsymbol{D}-\boldsymbol{W})\boldsymbol{Y})\\&=\operatorname{trace}(2\boldsymbol{Y}^{\mathrm{T}}\boldsymbol{L}\boldsymbol{Y})\end{aligned}$$

其中 $\boldsymbol{Y}$ 是降维后所有样本坐标值，$\boldsymbol{L}$ 就是原始样本的拉普拉斯矩阵，这也是 LE 模型名称的来源。

4. 优化函数求解

为了防止降维后所有样本有相同坐标值（此时所有$\|\boldsymbol{y}_i - \boldsymbol{y}_j\| = 0$，使得优化函数达到最小目标)，对上述的目标函数须加入约束条件$\boldsymbol{Y}^{\mathrm{T}}\boldsymbol{L}\boldsymbol{Y} = \boldsymbol{I}$。这种带约束条件的优化函数求解问题在上一小节 LLE 中刚刚遇到过，求解方法仍然是——拉格朗日乘子法。

由于又推导出了优化目标是$\min(\mathrm{trace}(2\boldsymbol{Y}^{\mathrm{T}}\boldsymbol{L}\boldsymbol{Y}))$的形式，因此只要对$\boldsymbol{L}$进行特征值分解并取最小的$d$个特征值对应的特征向量（$d$是降维目标空间中的维度数），就可直接求得$\boldsymbol{Y}$。

注意：实际上，很多模型的数学推导中都利用了这种将求解优化问题的拉格朗日乘子法转变为特征值分解后取若干特征向量的计算技巧，包括本章之前的 PCA、LDA 等模型。

5.5.3 调用介绍

在 sklearn.manifold 包中提供了 LLE 和 LE 模型的实现类 LocallyLinearEmbedding 和 SpectralEmbedding。其中 LocallyLinearEmbedding 不仅实现了第一小节描述的标准 LLE 模型，还可以通过它使用从 LLE 衍生的 Modified Locally Linear Embedding（MLLE）、Hessian Based LLE（HLLE）、Local tangent space alignment（LTSA）等模型。

这三种衍生模型只在 LLE 三步求解中的第二步与标准 LLE 有所不同。MLLE 和 HLLE 均是为了解决 LLE 近邻数大于目标维度数时产生的权重矩阵不能满秩的问题；而 LTSA 用局部几何分布替代权重矩阵表达近邻关系。

1. LocallyLinearEmbedding

本模型的关键初始化参数如下。

◎ n_neighbors：每个样本的近邻数量。

◎ n_components：降维后的目标维度数。

◎ reg：权重矩阵的正则化常数，标准 LLE 用它来防止权重矩阵不满秩。

◎ eigen_solver：特征值分解算法，与 Isomap 一样可以选择 arpack 或 dense。

◎ method：可以是 standard、hessian、modified 或 ltsa，对应标准 LLE 和它的三种衍生模型。

◎　neighbors_algorithm：近邻搜索算法，可选 brute、kd_tree、ball_tree。

在模型训练过后，可以读取如下属性获得降维结果。

◎　embedding_vectors_：低维空间中的样本向量，即降维结果。

◎　reconstruction_error_：降维后的权重矩阵重建误差。

◎　nbrs_：每个样本的近邻数。

2. SpectralEmbedding

本模型实现了拉普拉斯特征映射（LE）算法，重要的初始化参数如下。

◎　n_neighbors：每个样本的近邻数量。

◎　n_components：降维后的目标维度数。

◎　affinity：邻接矩阵 $\boldsymbol{W}$ 中样本相似度的计算方法，nearest_neighbors 表示只计算 n_neighbors 个近邻的相似度，rbf 用核函数计算全连接相似度，precomputed 需调用者自行计算。

在模型训练过后，可以读取如下属性获得降维结果。

◎　embedding_：低维空间中的样本向量，即降维结果。

◎　affinity_matrix_：邻接权重矩阵。

5.5.4　谱聚类

谱聚类（Spectral clustering）显然是一种聚类模型，它相对 K-means 具有计算代价低、适应非凸数据集等特点，特别适应于高维数据的聚类。

1. 谱聚类原理

没有将它写入第 4 章而是放在此处的原因是，其与拉普拉斯特征映射有密切联系，甚至只要一个等式就可以完全描述谱聚类的原理：

$$\text{谱聚类} = \text{拉普拉斯映射} + \text{K-means 聚类}$$

设目标聚类分组数是 k，谱聚类就是在对样本执行 LE 降维到 k 维后执行 centers=k 的

K-means 聚类。该过程决定了谱聚类的如下特性：

◎ 由于 LE 可以减少大量的样本数据维度数，使得在计算性能上比直接在高维数据上聚类的 K-means 等算法有大的飞跃。

◎ 由于流形学习是非线性算法，所以谱聚类能适应复杂的非凸数据集。

◎ 流形学习本身对噪声非常敏感，所以谱聚类对噪声的适应有所欠缺。

2. SpectralClustering 类

在 sklearn.cluster 包中的 SpectralClustering 类实现了谱聚类模型，类对象的初始化参数也基本是 LE 和 K-means 模型参数的组合。

◎ n_clusters：K-means 聚类目标分组数。

◎ eigen_solver：LE 特征值分解算法，与 Isomap 一样可以选择 arpack 或 dense。

◎ n_init：K-means 聚类次数。

◎ affinity：LE 模型中邻接权重矩阵值的计算方式。

◎ n_neighbors：LE 模型中每个样本近邻数量。

◎ assign_labels：由于 K-means 依赖于随机初始化的中心点，本参数允许在聚类步骤中使用除 K-means 外的 discretize 算法，它比 K-means 的结果更稳定。

在模型训练后，有两个属性可以读取。

◎ affinity_matrix_：LE 模型生成的邻接权重矩阵。

◎ labels_：给每个训练样本赋予的标签，即聚类结果。

3. 没有 predict ()函数

相比大多数聚类模型的一个缺点是，因为谱聚类需要将所有数据放在一起执行 LE 降维，谱聚类模型没有 predict()函数，即无法在训练后利用已有模型聚类新数据。

使用者只能通过 fit_predict()返回值获得训练数据的聚类结果，或者直接读取模型 labels_参数。

思考：局部嵌入类流形学习方法有哪些？它们的区别是什么？

5.6　流形学习之 t-SNE

T 分布随机近邻嵌入（t-distributed Stochastic Neighbor Embedding，t-SNE）是一种使用概率方法建模的流形学习降维方法。它诞生于 2008 年，是目前 scikit-learn 中最新的流形学习算法，最初用于高维数据的可视化。从降维效果来看 t-SNE 像是一个流形学习终极武器，而缺点是在训练时间方面较其他模型有明显增加。

5.6.1　用 Kullback-Leiber 衡量分布相似度

在学习了 Isomap、LLE、LE 等方法后，读者应该对流形学习的核心步骤有了明确的认识：

◎　用某种方式表达高维和低维空间样本之间的亲缘性（对 Isomap 来说是相似度，对 LLE 来说是近邻权重矩阵、对 LE 来说是拉普拉斯矩阵）。

◎　以亲缘性尽量不变为优化目标，用拉格朗日算子或特征值分解求得低维映射。

而在 t-SNE 中仍然是这样从定义亲缘性到求解优化目标的过程，略有不同的是：

◎　高维空间与低维空间的亲缘性并非用相同的方式表达，前者使用高斯分布，后者使用 t-分布。

◎　以两个分布的 Kullback-Leiber 散度最小作为优化目标。

1. 用高斯分布表达高维空间样本的亲缘性

在 t-SNE 中，假设每个样本都代表一个以它为均值 μ 的多元高斯分布，并且有各自的方差 σ。那么，可以用如下条件概率公式表达第 j 个样本与第 i 个样本所代表的高斯分布的亲缘程度：

$$p_{j|i} = \frac{\exp(-\| x_i - x_j \|^2 / 2\sigma_i^2)}{\sum_{k \neq i} \exp(-\| x_i - x_k \|^2 / 2\sigma_i^2)}$$

其中分子是 j 在 i 的高斯分布中出现的概率，分母是所有样本在 i 的高斯分布中出现的概率和。显然，该值越高表示对 i 来说 j 的亲缘性越高。相似的，可以计算出 $p_{i|j}$，也就是对 j 来说 i 的亲缘性程度。

在流形学习中，亲缘矩阵一定是一个对称矩阵，即该矩阵中的元素 $p_{ij}=p_{ji}$，该值可以用两个条件概率求平均值获得，即：

$$p_{ij}=p_{ji}=\frac{p_{i|j}+p_{j|i}}{2N}$$

这样，就可以利用训练样本计算出完整的概率亲缘性矩阵 $\boldsymbol{P}$ 了。

2. 高斯分布中的方差

在上述的亲缘性矩阵的计算过程中，假定了每个样本的高斯分布方差 σ 是已知的。而在实际的算法流程中，需要先确定每个样本所在高斯分布的 σ 值。该值可以是训练者手工输入的固定值，也可以由每个样本邻近的样本进行适配生成。适配过程是一个有监督的参数估计过程，不再详述。

需要理解的是，由邻近样本生成 σ 值的过程体现了 t-SNE 的流形学习本质，即每个样本有局部的欧几里得空间。

3. 用 t-分布表达低维空间样本亲缘性

t-分布又称为“学生 t-分布”，这是因为其发现者英国人 Willam S. Gosset 于 1908 年在发表论文时使用了笔名“Student”。在 t-SNE 中，低维空间的样本亲缘性用自由度是 1 的 t-分布计算。设 $\boldsymbol{y}$ 是低维空间样本，则有：

$$q_{ij}=\frac{f(\|\boldsymbol{y}_i-\boldsymbol{y}_j\|)}{\sum_{k\neq i}f(\|\boldsymbol{y}_i-\boldsymbol{y}_j\|)}，\text{其中 } f(t)=\frac{1}{1+t^2}$$

该值表达的含义与 p_{ij} 类似，都是在假设每个样本代表一个概率分布的情况下 i 与 j 的亲缘性，不同之处在于此公式中使用的概率分布是 t-分布。

4. Kullback-Leiber 散度

由于高维空间与低维空间使用不同的概率函数，因此两者的亲缘矩阵无法直接比较。而 Kullback-Leiber（简称 KL）散度是一种衡量两个概率分布相似程度的手段，KL 散度值越低表示两个概率分布越接近。因此 t-SNE 以最小化 KL 散度作为降维优化的目标：

$$J(\boldsymbol{Y})=\mathrm{KL}(\boldsymbol{P}\,\|\,\boldsymbol{Q})=\sum_{i,j}p_{ij}\log\frac{p_{ij}}{q_{ij}}$$

其中 $\boldsymbol{P}$ 矩阵是已知参数，$\boldsymbol{Y}$ 是低维空间样本向量，$\boldsymbol{Q}$ 矩阵可以由 $\boldsymbol{Y}$ 计算而来。因此以该目标作为损失函数，用梯度下降的方法就可以找到在 $\boldsymbol{Y}$ 为何值时 KL 值达到最小，此时的 $\boldsymbol{Y}$ 就是降维结果。

5.6.2　为什么是 t-分布

这里是本书第一次使用高斯之外的分布模型，为什么在 t-SNE 的低维空间没有使用万能的高斯分布呢？

其实在 t-SNE 出现之前有一个 SNE 模型，它正是一个在高维和低维空间都使用高斯分布的模型。而 t-SNE 由于将其中的低维分布改进为 t-分布而得名，此后在应用上完全替代了 SNE。为什么该项改动如此重要呢？这要从流形学习的拥挤问题说起。

1. 流形学习的拥挤问题（Crowding Problem）

在之前所有的流形学习模型中，几乎都围绕着“将近邻样本在高维空间中的相互距离尽量保持不变地映射到低维空间”这个主题而进行。这几乎是流形学习理论的出发点，以至于人们很少去想这个目标是否真的可行。

试想一个问题：在一个 d 维空间中，最多可以有多少个不同的点相互之间有相同距离？如果是 0 维空间，显然所有的点之间距离都相同，因此该值为 1；而 1 维空间是一条直线，则任取两个点后在该直线上再也找不到第三个点与它们的距离都相同；在 2 维空间中，任取一个等腰三角形后，再也没有第四个点能到三个顶点距离与三角形的边等长，如图 5-18 所示。

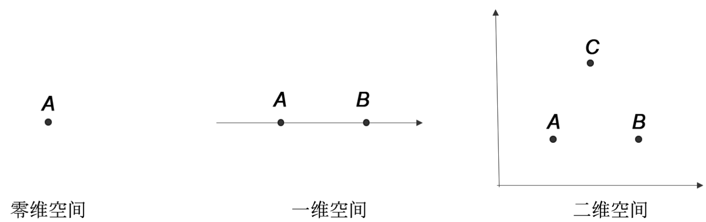


图 5-18　不同维度空间中的等距点

如此推理下去，此问题的答案也就显而易见：在 d 维空间中，最多可以有 d+1 个点相互之间的距离相同。也就是说，在高维空间等距的 n 个点，映射到维度数小于 n–1 的低维空间后根本不可能保持等距！这就是流形学习必须面对的拥挤问题。

2. t-分布解决拥挤问题

而在 t-SNE 中用 t-分布代替高斯分布拟合低维样本则在一定程度上解决了拥挤问题。如果读者对 t-分布不了解，此处只需知道关于它的两个特性：

◎ 与高斯分布有两个超参数（μ,σ）不同，t-分布只有一个超参数成为自由度，而在 t-SNE 中，t-分布的自由度固定为 1。

◎ t-分布是形态上与高斯分布类似的概率函数，但两侧的尾部比高斯分布高。其自由度越低，两侧尾部越高，由中心向两侧下降趋势越平缓；自由度越大，两侧尾部越低；当自由度趋于 ∞ 时，t-分布演变为标准高斯分布 $N(\mu=0,\sigma=1)$。

综合以上两点，t-SNE 中低维空间样本概率模型的尾部要显著高于高维空间样本的模型，其形状如图 5-19 所示。

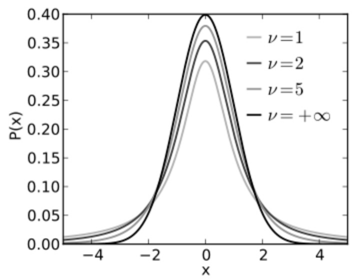


图 5-19　不同自由度的 t-分布（取自 Wiki 百科）

在图 5-19 中峰值最低、尾部最高的分布是自由度为 1 的 t-分布；峰值最高、尾部最低的是自由度为 ∞ 的 t-分布，也就是标准高斯分布。

在概率密度函数中，尾部高代表分布越分散，这样也就意味着用 t-分布建模的低维空间拉长了样本置信区间，因此合理地解决了所谓的“拥挤问题”。在 t-SNE 作者 Laurens van der Maaten 发表的第一篇关于 t-SNE 的论文 *Visualizing Data using t-SNE* 中对这项结论给出了详细的证明。

5.6.3　实战：使用 TSNE 类

在 scikit-learn 中对 t-SNE 模型进行封装的类是 sklearn.manifold.TSNE，重要参数如下。

◎ n_components：降维后的维度数。

◎ perplexity：用于调节流形中近邻的数量，通常取值范围在 5～50 之间，值越大在计算每个样本高斯分布的方差时使用的近邻越多。

◎ learning_rate、n_iter、min_grad_norm：使用梯度下降优化 KL 散度值时的学习速度、最大迭代次数、迭代停止阈值。

◎ metric：计算近邻距离的方法，缺省 euclidean 代表欧几里得空间距离。

◎ init：梯度下降开始点的选取方式，可以是 random、pca 或自定义低维样本值。

◎ method、angle：关于 t-SNE 重要衍生 Barnes-Hut t-SNE 算法的参数，该算法可以使梯度下降的时间复杂度从 $O(N^2)$ 降低到 $O(N \cdot \log N)$ 。

在完成模型训练后，可以读取如下模型属性。

◎ embedding_：降维后的低维空间样本值。

◎ kl_divergence_：梯度下降找到的最优 Kullback-Leibler 值。

◎ n_iter_：训练样本进行的梯度下降迭代次数。

TSNE 的训练、降维函数调用方式与之前的流形学习模型无异，此处不再举例。

思考：t-SNE 有什么优点？如何使用？

5.7　实战：降维模型之比较

在本章学了多种功能相近的降维模型后，从运算性能和降维结果上进行模型之间的实战比较是大有裨益的。

本节选取 scikit-learn 官网上 Jake Vanderplas 的流形学习分析案例，在保留功能的同时对原始代码做适当简化，并在其中加入 PCA 模型，呈现本章所有降维模型的整体感官。

本案例以将三维空间中的一个 S 状流形降到两维为目标，分别使用了 PCA、LDA、MDS、Isomap、LLE 及其两种衍生模型 LE 和 t-SNE。代码如下：

```
from time import time
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.ticker import NullFormatter
```

```
from sklearn import manifold, datasets
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA

Axes3D                                                   # 启用 matplot 3D

n_points = 1000                                          # 样本数量
X, color = datasets.samples_generator.make_s_curve(
                n_points, random_state=0)
n_neighbors = 10                                         # 流形学习近邻数量
n_components = 2                                         # 目标维度数

fig = plt.figure(figsize=(15, 8))

ax = fig.add_subplot(251, projection='3d')               # 在三维空间绘制原始数据
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=color, cmap=plt.cm.Spectral)
ax.view_init(4, -72)

methods = ['standard', 'ltsa', 'hessian']                # 定义三类 LLE 模型
labels = ['LLE', 'LTSA', 'Hessian LLE']

for i, method in enumerate(methods):             # 训练、显示 LLE 模型及其衍生模型
    t0 = time()
    Y = manifold.LocallyLinearEmbedding(n_neighbors, n_components,
                                        eigen_solver='auto',
                                        method=method).fit_transform(X)
    t1 = time()
    print("%s: %.2g sec" % (methods[i], t1 - t0))    # 用 t1-t0 计算训练时间
    ax = fig.add_subplot(252 + i)
    # 显示降维结果
    plt.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Spectral)
    plt.title("%s (%.2g sec)" % (labels[i], t1 - t0))
    ax.xaxis.set_major_formatter(NullFormatter())
    ax.yaxis.set_major_formatter(NullFormatter())
    plt.axis('tight')


# 初始化六种降维模型
estimators = [(manifold.Isomap(n_neighbors, n_components), "Isomap"),
            (manifold.MDS(n_components, max_iter=100, n_init=1), "MDS"),
```

```
            (manifold.SpectralEmbedding(n_components=n_components,
                                 n_neighbors=n_neighbors),
                                 "Laplace Eigenmaps"),
            (manifold.TSNE(n_components=n_components, init='pca',
                                 random_state=0), "t-SNE"),
            (PCA(n_components), "PCA"),
            (LDA(n_components=n_components), "LDA"),
            ]

# 训练、显示六种降维模型
for idx, (estimator_obj, estimator_name) in enumerate(estimators):
    # estimator_obj, estimator_name = estimator[0], estimator[1]
    t0 = time()
    if estimator_name=="LDA":                    # 如果是 LDA 模型，输入标签值
        Y = estimator_obj.fit_transform(X, (color).astype(int))
    else:
        Y = estimator_obj.fit_transform(X)
    t1 = time()
    # 用 t1-t0 计算训练时间
    print("%s: %.2g sec" % (estimator_name, t1 - t0))
    ax = fig.add_subplot(2, 5, 5+idx)
    # 显示降维结果
    plt.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Spectral)
    plt.title("%s (%.2g sec)" % (estimator_name, t1 - t0))
    ax.xaxis.set_major_formatter(NullFormatter())
    ax.yaxis.set_major_formatter(NullFormatter())
    plt.axis('tight')

plt.show()
```

上述代码首先用 make_s_curve()函数生成 *S* 状流形的 1000 个样本点，将三维特征数据保存在变量 X 中，将标签数据保存在 color 变量中。

在后续的模型降维训练中，只有有监督的 LDA 模型使用了标签数据，其他模型均只使用特征数据 **X**。而为了便于比较，在渲染结果时所有模型都使用了 color 参数。代码运行结果如图 5-20 所示。

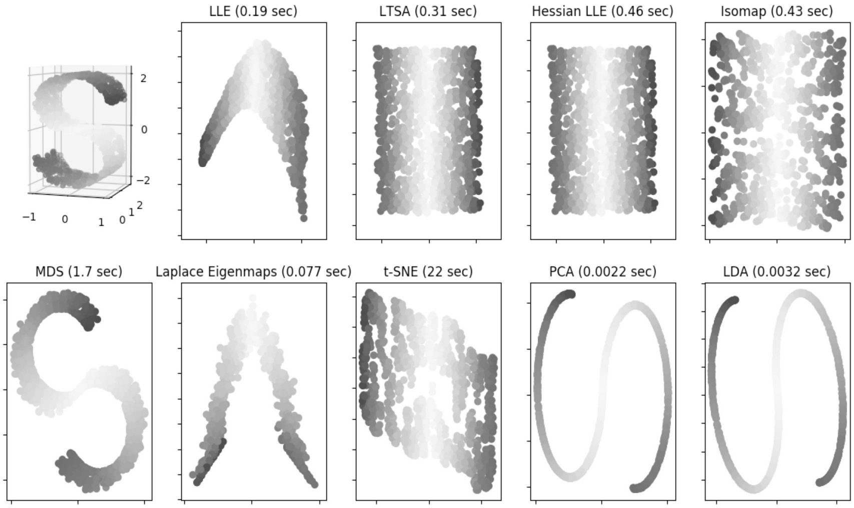


图 5-20　九种模型降维后的二维效果

图 5-20 的左上角子图是原始维度中的样本，其他九个子图是降维后的二维效果，每个结果子图的标题中显示了该模型的训练时间。可以发现：

◎ 原始样本像是一条二维彩带，卷曲在三维空间中。

◎ 从结果看，PCA 与 LDA 表现类似。都是找到了一个最能体现样本之间差异的切面进行三维到二维的映射，但都无法达到“展开”嵌入在外围空间中流形的目的。

◎ MDS 和流形学习模型都达到了“展开”彩带的效果，展开效果尤以 LTSA、Hessian LLE、Isomap、t-SNE 为佳。

◎ 从训练时间上看，PCA 最快，LDA 与其相近，之后是一众流形学习方法。因为需要计算所有样本的两两距离，MDS 也很慢。但最慢的是 t-SNE，耗时为倒数第二的 MDS 的十余倍，这是每个样本都需要拟合独立概率分布所致。

上述比较的时间结论基本代表了不同模型之间的内在差异，但降维效果在不同的场景中绝非定论，需要结合实际的数据分布、噪声情况仔细分析。

5.8　本章内容回顾

◎ 主成分分析 PCA：寻找最大方差切面的降维映射方法。

◎ 线性判别分析 LDA：是有监督的降维方法，有“类内方差最小化、类间方差最大化”的双重优化目标。

◎ 多维标度法 MDS：先将高维空间中的样本转换为两两之间的距离，然后以保留该距离信息作为目标进行降维。

◎ 除了传统的 MDS（也称 CMDS），non-metric MDS（NMDS）也有广泛应用。

◎ 流形（manifold=many fold）是嵌入外围空间中的由局部欧几里得空间组成的子空间。

◎ Isomap 的原理是用测地线计算距离，然后用 MDS 降维。

◎ 局部嵌入类流形学习以 LLE 为代表，在降维过程中仅关注近邻间的关系，后来衍生出了 Modified LLE、Hessian LLE、LTSA、LE 等方法。

◎ 谱聚类就是一种 LE+K-means 的聚类方法。运算速度快，但只能聚类训练数据，没有 predict()函数。

◎ t-SNE 是用高斯分布拟合高维空间样本、t-分布拟合低维空间样本的流形学习方法，效果通常好于其他流形学习模型，但计算速度非常慢。

第 6 章 隐马尔可夫模型

隐马尔可夫模型（Hidden Markov Model，简称 HMM）是一个带有隐含状态的基于统计方法的马尔可夫模型，其在有状态的智能系统中有丰富的应用空间，比如金融统计、语音输入识别、越来越智能的汉语输入法等。隐马尔可夫模型以贝叶斯网络作为数据框架，用 Baum-Welch 算法进行无监督训练，通过 Viterbi 算法计算隐含状态。

本章将学习 HMM 的应用场景、算法原理和开发过程。

◎ HMM 中的状态与概率：学习 HMM 的可见状态、隐含状态、转换概率和表现概率。

◎ HMM 可以解决哪些问题：提出三种可以解决的 HMM 问题。

◎ 算法原理：介绍前后向算法、Viterbi 和 Baum-Welch 等算法基本原理。

◎ hmmLearn 入门：安装和简单使用。

◎ hmmLearn 应用：解决三大 HMM 问题，包括对 Viterbi 和 Baum-Welch 等算法各种参数的调试等。

◎ 案例分析：解读如何用 HMM 进行股票涨跌预测。

6.1 场景建模

本章通过一个现实生活问题引导读者学习 HMM 中的几个关键概念，为理解模型算法和开发应用打下基础，同时也启发读者思考如何将 HMM 应用到自己的业务/产品系统中。

6.1.1 两种状态链

隐马尔可夫模型围绕两组状态序列进行建模，它们分别是显示状态链和隐藏状态链。

1. 马尔可夫模型

说起隐马尔可夫的历史，其来源于 1906 年俄罗斯科学家安德烈·马尔可夫提出的离散随机过程链，即马尔可夫模型。后来柯尔莫果洛夫于 1936 年将其一般化到可数的无限状态空间的状态表示。虽然这些词说起来比较拗口，但其实可以通过以下几条特点理解马尔可夫模型。

◎ 它表示的是一个有顺序的状态转换模型，可以用 t_1、t_2、$t_3 \ldots t_n$ 的方式表示这个序列。

◎ 序列中的每个结点可以取值为若干状态之一（或是连续区间中的一个值），但一个结点不可以有多个状态。

◎ 这个状态序列可以是无限长的。

◎ 该模型可以用来沿着这个状态链“以概率的形式根据历史状态预测未来状态”。

◎ 每次状态的预测只以其最近的一个状态作为依据，比如预测 t_5 的状态时只用 t_4 的状态作为依据，而不去考虑 t_1、t_2 或者 t_3 的状态。

天气状态链是最简单的马尔可夫模型。每一天是一个结点，每个结点的状态可以取为

晴、多云、雨等，各状态之间以一定概率互相转换（比如，第一天为晴天，那么第二天也可能有 60%的概率是晴天，又有 30%的概率是阴天，还有 10%的概率是雨天）。连续若干天的天气情况就组成了一个马尔可夫链，天气状态链如图 6-1 所示。



图 6-1　天气状态马尔可夫链

生活中的很多场景可以简化为马尔可夫模型，下面举两个例子。

（1）股票状态链：一个股票每一天的走势都是一个结点，每一天的状态可以分为大涨、小涨、平、小跌、大跌等，连续若干天的该股股票走势就是一个马尔可夫链。

（2）文本链：这个稍微有些抽象，一段文本的每个字可以看成一个结点，每个字的取值范围就是状态集（对于中文来说，就是新华字典中的约 7000 个汉字），那么每个词组、每句话、每篇文章都是一个马尔可夫链。

2. 隐马尔可夫模型

在 20 世纪 60 年代，L.E.Baum 等人在一系列论文中扩充了马尔可夫模型。他们建立了一个由隐藏层和可见层组成的两层模型，底层隐藏层由马尔可夫状态链组成，可见层由普通状态结点组成。因此，该模型被命名为隐马尔可夫模型，训练该模型的算法是 Baum-Welch 算法。

理解隐马尔可夫模型的关键点如下：

◎　显示层的结点状态是可以直接获取的。

◎　作为隐藏层的马尔可夫链的各结点状态不直接可见，需要用可见层的状态去推测。

生活中有很多这样的场景，同样以天气状态进行举例。假设有这样一个问题，远在另一个城市上大学的儿子每天通过邮件向你汇报他今天做的最多的事情是什么。这些事情可能是这三项之一：打球、读书、访友。那么在这种场景下，你如何推测儿子所在城市的天气情况呢？

这个问题看似有些无厘头，但其实人的活动确实可能和当地的天气有关，隐马尔可夫模型就是用来对这种“在一定程度上相关，但又不是绝对相关的两种事件”进行建模分析

的。在这个问题中，儿子的活动类型构成了模型的可见层（因为儿子每天会向您汇报这个情况），天气状况成了隐藏层（儿子不向您汇报，您需要猜测这个情况）。假设儿子的活动有如下规律：在晴天更可能打球，在阴天更可能访友，在雨天更可能读书，那么实际上的天气情况可能如图 6-2 所示。

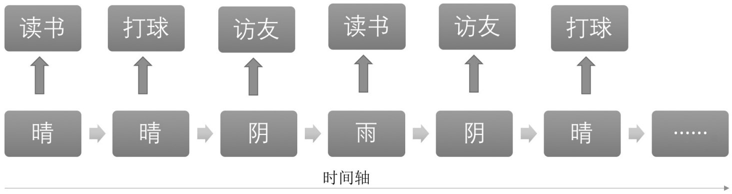


图 6-2　隐藏天气状态的两层状态链

在图 6-2 中除第一组结点外，其他结点都符合儿子的活动规律。注意，因为显示层与隐藏层之间的状态不是绝对的映射关系，所以图 6-2 显示的天气隐藏层状态只是一种可能的状态，不是绝对可靠的事件序列。

这就是隐马尔可夫模型中的重要概念：顺序序列上的可见层链和隐藏层链。

读者可能会想，这么简单的模型怎么能解决金融数据预测、语音识别这样复杂的问题呢？原因就是接下来讲的模型中的两种概率起的关键作用。

6.1.2　两种概率

在前面一节介绍 HMM 中的状态链时，多次提到了“可能”“一定程度”“非绝对”等字样，这些抽象概念最终是通过模型中的两种概率表来起具体作用的。

两种概率分别在图 6-2 中的状态变换过程中（即两种方向的箭头上）起作用，下面分别讲解。

1. 输出概率

图 6-2 中的若干垂直方向由下至上的箭头代表了 HMM 中的输出概率（emission probability），它的含义是隐含层的每一种状态的结点都以一定的概率表现为可见层的结点状态。表 6-1 是对应于图 6-2 的一个输出概率表的例子。

表 6-1　输出概率表

序号	隐藏层状态	可见层状态	输出概率
1	晴	打球	0.4
2	晴	读书	0.3
3	晴	访友	0.3
4	阴	打球	0.2
5	阴	读书	0.3
6	阴	访友	0.5
7	雨	打球	0.1
8	雨	读书	0.8
9	雨	访友	0.1

可以把隐藏层状态和可见层状态看成一组因果关系：隐藏层是因，可见层是果。解读表 6.1 的含义：晴天的情况下，有 40%的可能打球，有 30%的可能读书，有 30%的可能访友；阴天情况有 20%的可能打球，……，该表有如下特点：

◎ 每一种隐藏层状态对各可见层状态的输出概率总和必定为 1。比如表 6.1 中的 1~3 行的输出概率总和为 0.4 + 0.3 + 0.3 = 1，4～6 行、7～9 行的总和都各自为 1。

◎ 可见层每种状态的“被输出概率”总和可以是任意值，比如“打球”状态的被输出概率总和为 0.4 + 0.2 + 0.1 = 0.7。

2. 转换概率

图 6-2 中水平方向箭头表示的隐藏层马尔可夫链上的变换被称为转换概率（transition probability），它反映的是隐藏层内部各状态之间的固有转换规律。如表 6-2 所示是对应于图 6-2 的一个转换概率表的例子。

表 6-2　转换概率表

序号	*T* 时刻隐藏层状态	*T*+1 时刻隐藏层状态	转换概率
1	晴	晴	0.7
2	晴	阴	0.2
3	晴	雨	0.1
4	阴	晴	0.3
5	阴	阴	0.5

续表

序号	*T* 时刻隐藏层状态	*T*+1 时刻隐藏层状态	转换概率
6	阴	雨	0.2
7	雨	晴	0.3
8	雨	阴	0.4
9	雨	雨	0.3

读者应该还记得本节开始描述的马尔可夫链的特点之一是“每次状态的预测只以其最近的一个状态作为依据”，因此在表 6-2 描述的转换概率因果关系中，“因”只有一列状态即“*T* 时刻隐藏层状态”，而根据它转换成的结果为“*T*+1 时刻隐藏层状态”。表 6-2 可以解读为，如果第一天是阴天，那么第二天有 50%的可能也是阴天（第 5 行），另有 30%的可能是晴天（第 4 行），还有 20%的可能是下雨（第 6 行）；如果第二天是晴天，那么第三天有 70%的可能也是晴天，另有 20%的可能是阴天，……

有时，转换概率表也可以用状态图的方式表达，如图 6-3 所示。

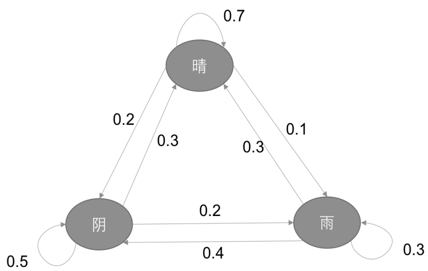


图 6-3　转换概率状态图

在图 6-3 中，九个状态转换连线分别对应图 6-2 的九行数据，状态图与概率转换表是完全可以互相表征的。

6.1.3　三种问题

至此已经学习了隐马尔可夫模型的静态与动态结构，那么基于 HMM 的场景可以解决哪些问题呢？

在模型中的隐藏层状态、可见层状态、输出概率、转换概率等几种数据中，通常有一些是未知的，而 HMM 的目的就是找出这些未知数据。通常可以把这些问题归结为三

种类型：

◎ 问题一，状态问题。已知模型的所有参数（隐藏层状态类型、可见层状态类型、输出概率表、转换概率表），求任意一个可见层状态链产生的几率。在之前描述的学生活动/天气模型中，这类问题包括：求连续三天读书的概率是多少？求第一天访友、第二天打球的概率是多少？

◎ 问题二，解码问题，隐藏层状态猜测。已知模型的所有参数，并且知道某一组可见层的状态链，求这些可见状态下对应的隐藏状态链是什么。在学生活动/天气模型中，这类问题类似于，已知学生连续四天的活动分别为读书、读书、打球、访友，求这四天的天气情况是什么？

◎ 问题三，训练模型参数。已知隐藏层状态数量（本节天气模型中为晴、阴、雨三种）、可见层状态数量（读书、访友、打球三种）、足够长的可见状态链，求模型中的输出概率表和转换概率表。仅知之前的例子模型即图 6-2 中第一行的一系列状态，求表 6-1 和表 6-2 中的概率值。自然，已知的可见层状态链越长，模型参数的推测也越准确。

另外，问题一与问题二也可以各自与问题三结合起来解决现实问题，即：

◎ 已知一系列的可见层状态，求某一段可见层的概率。即先求问题三，再用得到的模型求解问题一。比如，已知学生两个月的活动状态，求他在接下来的三天分别选择读书、读书、访友的可能性。

◎ 已知一系列的可见层状态，求它们下面对应的隐藏层状态。即先求问题三，再用得到的模型求解问题二。比如已知学生两个月的活动状态，求这两个月当地的天气如何。

用直觉就可以感觉到，凡是涉及第三种问题的情况，都需要有足够长的可见模型用于训练模型参数，否则求出的模型不可能准确。比如，如果仅知道学生连续三天的活动为读书、访友、访友，那么要求其推测模型参数显然勉为其难。

思考：隐马尔可夫模型围绕哪两组状态序列进行建模？

6.1.4 hmmLearn 介绍

了解 HMM 的场景建模后，就可以开始小试牛刀，进行基于隐马尔可夫问题的应用开

发了。

在 Python 中有一个专门用于解决隐马尔可夫模型问题的框架 hmmLearn。它最开始是 scikit-learn 框架的一部分，因为相对其他问题的独立性被剥离出来成为单独的项目。

思考：之前的章节已经学过 scikit-learn，为什么说 HMM 相对 scikit-learn 中的其他模型是独立的？scikit-learn 中的模型功能都比较单一，比如分类、回归、聚类等，但 HMM 模型却有三种常见问题，综合了监督学习和非监督学习的特点。

1. 安装

像其他 Python 框架一样，hmmLearn 的安装非常简单。打开终端命令行，输入如下 pip3 命令即可：

```
#pip3 install hmmlearn
```

2. 功能纵览

根据从隐藏层状态到可见层状态输出概率的不同形式，hmmLearn 提供了如下三种 HMM 模型。

◎ MultinomialHMM：适合用于可见层状态是离散类型的情况。所谓离散类型可以理解为编程中的枚举类型，他们的数量有限、值与值之间没有数量上的“多与少、大与小”的关系。表 6-1 的输出概率表即为这种类型，因为读书、打球、访友三种状态数量有限，同时相互之间没有大小顺序关系。MultinomialHMM 模型的构造函数原型如下：

```
class hmmlearn.hmm.MultinomialHMM(
n_components=1, startprob_prior=1.0, transmat_prior=1.0,
algorithm='viterbi', random_state=None, n_iter=10, tol=0.01,
verbose=False, params='ste', init_params='ste')
```

◎ GaussianHMM：适合用于可见层状态是连续类型且假设输出概率符合 Gaussian 分布的情况。所谓连续类型是指状态值较多，而且值之间有大小关系。比如，在金融数据模型中常把某金额/某数量作为可见层状态，此时可以用 Gaussian 分布。GaussianHMM 模型的构造函数原型如下：

```
class hmmlearn.hmm.GaussianHMM(
n_components=1, covariance_type='diag', min_covar=0.001,
```

```
    startprob_prior=1.0, transmat_prior=1.0, means_prior=0, means_weight=0,
    covars_prior=0.01, covars_weight=1, algorithm='viterbi',
    random_state=None, n_iter=10, tol=0.01, verbose=False, params='stmc',
    init_params='stmc')
```

思考：这里的连续概念与严格意义上的数学连续有所区别，虽然金融市场金额／数量等的取值范围在数学概念上不是连续（比如金额可能是 12.78、53.782，但不会是 3.4512984 这样的数）的，但由于它们是有大小关系的序列，也可以使用 Gaussian 分布。

◎ GMMHMM：适合用于隐藏层状态是连续类型且假设输出概率符合 GMM 分布（Gaussian Mixture Model，混合高斯分布）的情况。混合高斯分布可以想象为有多个峰值的分布。举个例子，如果某股票市场经常暴涨暴跌，而平盘或微幅波动的情况很少，则可以选择 GMMHMM 模型。GMMHMM 模型的构造函数原型如下：

```
class hmmlearn.hmm.GMMHMM(
n_components=1, n_mix=1, min_covar=0.001, startprob_prior=1.0,
 transmat_prior=1.0, weights_prior=1.0, means_prior=0.0, means_weight=0.0,
 covars_prior=None, covars_weight=None, algorithm='viterbi',
 covariance_type='diag', random_state=None, n_iter=10, tol=0.01,
 verbose=False, params='stmcw', init_params='stmcw')
```

注意：这里已经不是本书第一次用到 Multinomial 和 Gaussian 的概念，对其不熟悉的读者可回顾本书第 3、4 章。

三个模型的所有构造函数的参数都有默认值，即使对参数含义不甚了解也可以开始使用 hmmLearn 进行编程。但要真正调试出适配自己场景的模型，还需要对各关键参数熟练掌握，本章后续将会逐步揭秘这些参数。

3. 小试牛刀

现在我们开始用 hmmLearn 实现本节中的经典案例：学生活动/天气状况 HMM 模型。这里的目标是已知一组模型参数，然后对给定学生活动（可见层）预测当时可能的天气情况（隐藏层），也就是解决第二类问题——隐藏层状态猜测。

首先按照表 6-1 的输出概率创建 Numpy 二维数组：

```
import numpy as np

emission_probability = np.array([
  [0.4, 0.3, 0.3],  # 晴的隐藏状态时：打球概率为 0.4，读书概率为 0.3， 访友概率为 0.3
```

```
    [0.2, 0.3, 0.5],  # 阴的隐藏状态时：打球概率为 0.2，读书概率为 0.3， 访友概率为 0.5
    [0.1, 0.8, 0.1]   # 雨的隐藏状态时：打球概率为 0.1，读书概率为 0.8， 访友概率为 0.1
  ])
```

录入表 6-2 的转换概率表：

```
  transition_probability = np.array([
    [0.7, 0.2, 0.1],          # 晴->晴概率为 0.7，->阴概率为 0.2，->雨概率为 0.1
    [0.3, 0.5, 0.2],          # 阴->晴概率为 0.3，->阴概率为 0.5，->雨概率为 0.2
    [0.3, 0.4, 0.3]           # 雨->晴概率为 0.3，->阴概率为 0.4，->雨概率为 0.3
  ])
```

定义隐藏层各状态的初始概率：

```
  start_probability = np.array([0.5, 0.3, 0.2])
```

这个初始概率在本章中是第一次提到，它的含义是隐藏层状态链中第一个结点的状态概率分布（因为第一个结点没有前结点，所以需要靠初始概率进行设置）。它与转换概率表组合在一起完整地描述了隐藏层所有结点的状态概率分布。在以上代码中，第一个结点晴天的概率最高（占 5 成），其次是阴天（占 3 成），最低是雨天（占 2 成）。

建立 MultinomialHMM 对象，并将概率表作为已知条件传给模型：

```
  model = hmm.MultinomialHMM(n_components=3)   #定义有 3 种可见层状态
  model.startprob_ = start_probability
  model.transmat_ = transition_probability
  model.emissionprob_ = emission_probability
```

至此，一个已知参数的隐马尔可夫模型已经定义完毕。可以直接用来根据学生行为预测天气状态了：

```
  observe_chain  =  np.array([0,2,1,1,1,1,1,1,2,2,2,2,2,2,2,2,1,0]).reshape
(-1, 1)
  print(model.predict(observe_chain))
```

代码已经全部给出，运行结果如下：

```
  [0 0 2 2 2 2 2 2 1 1 1 1 1 1 1 1 0 0]
```

在 hmmLearn 中可见层和隐藏层的状态分别从 0 开始编号。对照 emission_probability 矩阵中数值的顺序，我们知道对 observe_chain 中的值来说，0 代表打球，1 代表读书，2 代表访友；对预测的隐藏状态来说，0 代表晴，1 代表阴，2 代表雨。

再仔细一一对照可见层和隐藏层状态。

```
[0,2,1,1,1,1,1,1,2,2,2,2,2,2,2,2,1,0]      # 可见层
[0 0 2 2 2 2 2 2 1 1 1 1 1 1 1 1 0 0]      # 隐藏层
```

可见总体上符合输出概率表中的配对：晴天时打球，阴天访友，雨天读书。但仍然有两天例外：即第二天访友时的天气被预测为晴天，倒数第二天的读书也被预测为晴天。此时再回顾初始概率和转换概率的值，可知这是因为晴天本身在隐藏层所有状态中占比较大，直觉就会发现结果非常合理，这就是隐马尔可夫模型的魅力。

思考：hmmLearn 提供了哪三种 HMM 模型？

6.2 离散型分布算法与应用

虽然已经可以用 hmmLearn 进行 HMM 开发，但可能读者还会好奇框架到底是用什么方法解决 HMM 的三大问题的。本节用形象化的方式介绍这些算法的奥秘，并给出在 hmmLearn 中相关的函数。

6.2.1 前向算法与后向算法

解决可见层状态链可能性评估的算法可以是暴力求解、前向算法（Forward algorithm）或后向算法（Backward algorithm），后两者也合称为 forward-backward algorithm。

1. 暴力求解

所谓可能性问题就是求解一个固定可见状态链出现的概率，这个概率最直接的求解方法步骤如下：

- ◎ 计算每一种隐藏状态链的出现概率。
- ◎ 针对每一种隐藏状态链，计算在其出现的情况下给定可见状态链的出现概率。
- ◎ 所有隐藏状态链下计算的给定可见状态链概率求和即得到题解。

下面用形象化的例子图解这个过程，假设在活动/天气模型中给出的可见层状态是：读书→打球→访友→读书。如图 6-4 所示是一种可能的隐藏状态链（阴→晴→雨→阴）。

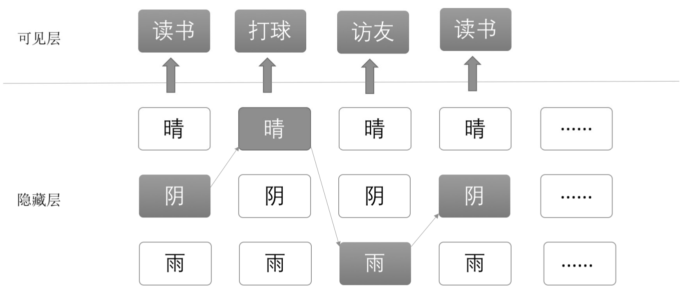


图 6-4　暴力求解之隐藏状态链一

图中浅色背景的结点是每个隐藏层结点未被选中的状态，对照 6.1.2 节中对初始概率和转换概率表 6-2 的设定，这条隐藏状态链出现的概率是：

```
0.3(第一个结点是阴的初始概率)*
0.3(转换概率表中阴转晴的概率)*
0.1(转换概率表中晴转雨的概率)*
0.4(转换概率表中雨转阴的概率) = 0.0036 （图 6-4 的隐藏状态链的总概率）
```

对照输出概率表 6-1 的设定，在该条隐藏状态链下可见层状态链的出现概率是：

```
0.3(阴时读书的概率)*
0.4(晴时打球的概率)*
0.1(雨天访友的概率)*
0.3(阴天读书的概率) = 0.0036 (在图 6-4 的隐藏状态链下该可见层的出现概率)
```

因此，图 6-4 对给定可见层链（读书→打球→访友→读书）贡献的概率是：

```
0.0036（隐藏层出现的概率）*
0.0036（该隐藏层下的可见层输出概率） = 0.00001296 （图 6-4 的贡献概率）
```

如图 6-5 所示是另一种可能的隐藏状态链。

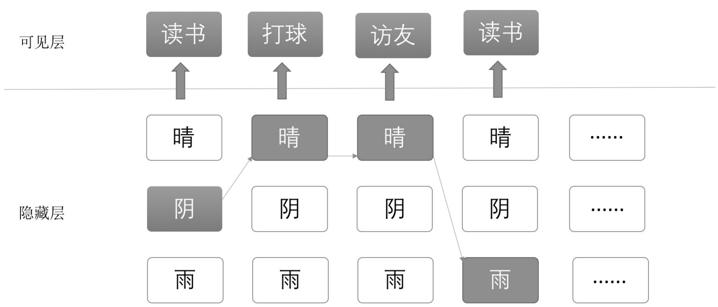


图 6-5　暴力求解之隐藏状态链二

不再赘述，有兴趣的读者可以自行计算图 6-5 对给定可见层状态的贡献概率。

设隐藏状态数量为 N（本例中为 3），结点数量为 T（本例中为 4），对它们进行排列组合，可以知道诸如图 6-4 和图 6-5 的情况一共有 N^{T} 种（本例中即 pow(3, 4)=81 种）。对所有这些情况的贡献概率相加，即可得到最终的可见层链的概率。

2. 前向算法

暴力求解法似乎已经可以完美地解决 HMM 的问题，那么为什么还会有其他算法呢？原因就是暴力求解太过耗时！可以找到其他方法得到和它一样的解，同时时间复杂度又大大降低。

经过前面的分析，暴力求解法的时间复杂度是 $O(T \cdot N^{\mathrm{T}})$，它是指数级的。所谓前向算法，就是用动态规划（dynamic programming）的方法改造计算过程，使其最终的时间复杂度能够达到 $O(T \cdot N^2)$，是多项式级别的算法。

提示：动态规划的核心是一种以空间换时间的编程方法，对其细节感兴趣的读者可以参阅 Thomas H.Cormen 等编写的经典书籍《算法导论》。

依然延续前面的图解举例，回顾图 6-4 和图 6-5 可以发现其实这两种隐藏层状态有一部分是“共同路径”，即它们的前两个结点状态相同（都是阴→晴）。那么在计算过程中，就可以先计算这两种状态的概率，再计算其后结点各自出现的概率，如图 6-6 所示。

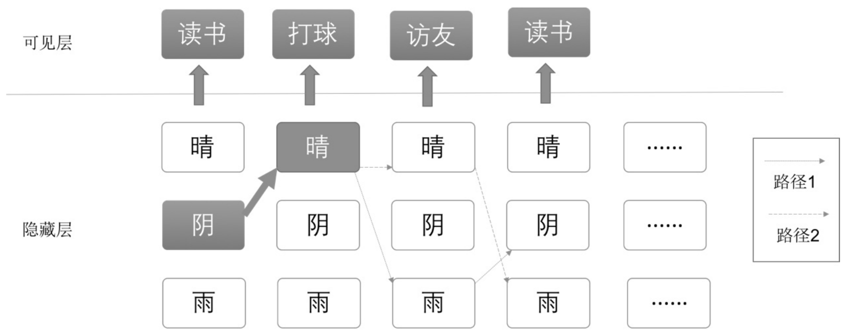


图 6-6　前向算法图解

在图 6-6 中，先计算前两个隐藏结点是阴→晴的情况下的概率，即：

```
0.3(第一个结点是阴的初始概率)*
0.3(转换概率表中阴转晴的概率) = 0.09 (前两个结点是阴→晴的概率)
```

这样在计算路径 1 时就可以直接利用该数值了：

```
0.09 * 0.1(转换概率表中晴转雨的概率) = 0.009 (前三个结点是阴→晴→雨的概率)

0.009*0.4(转换概率表中雨转阴的概率) = 0.0036(四个结点是阴→晴→雨→阴的概率，与暴力求解所得一致)
```

上述两行之所以要分开计算，是因为像第二列隐藏层可能结点的状态相同，0.009 这个第三列的数值也需要存储起来，供第四列结点在计算时使用。

因此，前向算法的步骤可以总结为：

◎　通过初始概率读取第一列隐藏层所有可能状态的概率值。

◎　根据第一列所有的状态概率值计算第二列隐藏层所有的可能状态概率值。

◎　用第二列的概率值计算第三列的概率值。

◎　以此类推直至最后一列。

◎　然后计算这个过程中产生的所有路径的输出概率，并求和。

3. 后向算法

后向算法与前向算法非常相似，也是用动态规划的方法按照层次分别计算每一列隐藏结点可能状态的概率值。两者区别是前向算法从第一个结点开始向前推导，后向算法从最后一个结点向后推导。后向算法的过程为：

◎　计算隐藏层最后一个结点（设为 t 时刻）所有可能状态的后向概率值。

◎　根据所有 t 的状态后向概率值计算 t-1 时刻隐藏层所有可能状态的后向概率值。

◎　用 t-1 时刻的概率值计算 t-2 层的概率值。

◎　以此类推直至第一列。

◎　然后计算这个过程中产生的所有路径的输出概率，并求和。

其时间复杂度也是 $\mathrm{O}(T \cdot N^{\mathrm{T}})$。其中出现了“后向概率”这个概念，其实它就是从当前结点开始向时间轴方向观测的概率值。

6.2.2 MultinomialNB 求估计问题

在 hmmLearn 中所有的 HMM 模型类都有一个.score()成员函数，用于为给定模型的某个可见状态计算可能性，其函数原型是：

```
    score(X, lengths=None)
    Compute the log probability under the model.
    Parameters:
    X : array-like, shape (n_samples, n_features)
       Feature matrix of individual samples.
    lengths : array-like of integers, shape (n_sequences, ), optional
       Lengths of the individual sequences in X. The sum of these should be n_samples.
    Returns:
    logprob : float
       Log likelihood of X.
```

代码中的参数 X 是一个可见层状态的 Numpy 数组，输出一个 log 概率值。所谓 log 概率值是对原始概率进行 ln（即 $\log_e$）计算后获得的值。比如：

```
    >>> observe_chain = np.array([0,2,1,1,1,1,1,1,2,2,2,2,2,2,2,2,1,0]).reshape(-1, 1)
    >>> print(model.score(observe_chain))
    -8.9
```

假设该模型输出 ln 的概率是−8.9，则其真实的概率值应该是 $e^{-8.9}$≈0.000136=0.136%。

考虑如图 6-7 所示的 ln 函数曲线图，由于概率值的取值范围是 0~1，所以.score()函数的输出范围应该在−∞～0 之间。

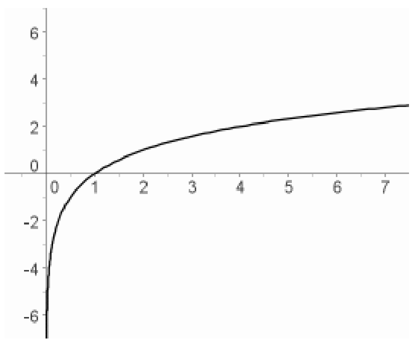


图 6-7 ln 函数曲线图

在图 6-7 中，ln 函数在 x 越接近于 0 的时候，y 值会更快地加速趋于 0。要将.score()的 ln 概率值转换为真实概率值，可以使用 Numpy 的 exp 函数，比如：

```
>>> import numpy as np
>>> np.exp(-10.7)
2.254493791321221e-05
```

说明：hmmLearn 不直接返回概率值而是使用 log 概率值的原因是：计算机对浮点数的有效位数有限制，而 log 概率能更好地利用有限的浮点数有效位。

6.2.3　Viterbi 算法

维特比算法（Viterbi Algorithm）用来在给定 HMM 模型参数和一个可见状态链时，找出最可能的隐藏状态链。

维特比算法的过程可以分为两步：

◎　正方向推算，用动态规划的方法从第一个隐藏结点开始逐步计算每个可能状态：

i．计算本状态出现的概率，设为δ。

ii．最可能达到本状态的前一个结点的状态，设为 Ψ。

◎　反方向推理，计算到最后一个结点时，概率最大的那个状态就是最后一个隐藏结点的被推测状态。然后逐步向时间轴反方向推导，逐个找出最可能的隐藏结点。

仍然利用前面的活动/天气模型进行举例说明，假设现有如图 6-6 所示的可见状态链：读书→打球→访友→读书。

1．计算第一个隐藏结点各状态的δ与 Ψ

第一个可见结点是读书，隐藏结点是三个可能状态的δ与 Ψ：

```
δ1(0) = 0.5(晴的初始概率) * 0.3（晴天读书的概率） = 0.15
δ1(1) = 0.3(阴的初始概率) * 0.3（阴转晴的概率） = 0.09
δ1(2) = 0.2(雨的初始概率) * 0.8（晴转晴的概率） = 0.16
Ψ1(0) = Ψ1(1) = Ψ1(2) = 0                # 其实第一个结点Ψ无意义，赋初始值为 0
```

$\delta_1(0)$、$\delta_1(1)$、$\Psi_1(0)$等符号的含义：$\delta_1(0)$是在第一个结点取状态 0 即晴天时的概率；$\delta_1(1)$是在第一个结点取状态 1 即阴天时的概率；$\Psi_1(0)$是第一个结点取状态 0 即晴天时最有可

能的前一个状态。

2. 计算第二个隐藏结点各状态的δ与Ψ

从第二个结点开始，δ的计算需要针对前一个可能值取最大可能状态。在本例中第二个结点的可见状态是打球，用表 6-3 描述$\delta_2(0)$和$\Psi_2(0)$的计算过程。

表 6-3　$\delta_2(0)$与$\Psi_2(0)$值计算示例

步骤	步骤	计算
1	第一个隐藏结点是晴、并且第二个结点也是晴的概率	0.15 ($\delta_1(0)$) × 0.7(转换概率) = 0.105
2	第一个隐藏结点是阴、并且第二个结点是晴的概率	0.09 ($\delta_1(1)$) × 0.3(转换概率) = 0.027
3	第一个隐藏结点是雨、并且第二个结点是晴的概率	0.16 ($\delta_1(2)$) × 0.3(转换概率) = 0.048
4	取以上三条计算的最大值并乘以晴→打球的输出概率作为$\delta_2(0)$	$\delta_2(0)$ = 0.105 × 0.4 = 0.042
5	取使$\delta_2(0)$达到最大的第一结点状态作为$\Psi_2(0)$	$\Psi_2(0)$ = 0　（即前一个结点是 0 时δ2(0)最大）

表 6-3 描述的是结点 1 在晴天状态下的δ与Ψ值，算法还需用这些步骤完成阴、雨两种其他状态的δ与Ψ值计算，即$\delta_2(1)$、$\Psi_2(0)$、$\delta_2(2)$、$\Psi_2(2)$。

3. 逐步计算所有隐藏结点的δ与Ψ

像第二步一样逐步完成第三、四个隐藏结点δ与Ψ的计算。这里不再重复叙述，读者可以自行演算。

4. 获得最后一个结点的隐藏层状态

假设最后一个结点的δ值为：

```
δ4(0) = 0.000015
δ4(1) = 0.0000038
δ4(2) = 0.0000129
```

此刻即可得到一个结论：最后一个结点的最可能隐藏状态是 0 即晴天。

5. 反方向推理

有了最后一个结点的隐藏状态，就可以用Ψ值逐个向前推理所有隐藏结点的状态了。假设经过前面的计算获得了如下Ψ值：

```
Ψ2(0) = 2
Ψ2(1) = 1
Ψ2(2) = 2

Ψ3(0) = 0
Ψ3(1) = 0
Ψ3(2) = 2

Ψ4(0) = 1
Ψ4(1) = 0
Ψ4(2) = 2
```

那么推导隐藏状态的过程为：

◎　因为 Day4 的推测状态是 0，并且 $\Psi4(0) = 1$，所以 Day3（倒数第二天）的推测状态是 1。

◎　又因为 Day3 的推测状态是 1，并且 $\Psi3(1) = 0$，所以 Day2 的推测状态是 0。

◎　又因为 Day2 的推测状态是 0，并且 $\Psi2(0) = 2$，所以 Day1 的推测状态是 2。

至此已经完成维特比算法的推导，得到所有结点的隐藏状态推测：2、0、1、0。

注意：本节中第三、四结点的δ与Ψ数值仅是为了便于讲解算法过程的假设值，如自行演算会发现本案例的最终结果应该是[0, 0, 0, 0]，但这不利于讲解，所以略作调整。

6.2.4　MultinomialNB 求解码问题

在 hmmLearn 中可以直接调用几个 HMM 模型的.predict()函数用 Viterbi 算法进行隐藏状态推测，函数原型如下：

```
    predict(X, lengths=None)
    Find most likely state sequence corresponding to X.
    Parameters:
    X : array-like, shape (n_samples, n_features)
       Feature matrix of individual samples.
    lengths : array-like of integers, shape (n_sequences, ), optional
       Lengths of the individual sequences in X. The sum of these should be
n_samples.
    Returns:
```

```
    state_sequence : array, shape (n_samples, )
        Labels for each sample from X.
```

其输入参数 X 是一个可见层状态的 Numpy 数组，lengths 参数用来说明 X 中每条可见链的长度。X 与 lengths 参数结合使得.predict()参数一次可以预测多条可见链的隐藏层，下面举例说明。

1. X 由 1 个可见链构成，每个可见结点有 1 个特征

这是一种最简单的情况，也就是本章一直举例的活动/天气模型情况：

```
X= np.array([0,2,1,1,1,1,1,1,2,2,2,]).reshape(-1,1)
model.predict(X, [11,])
```

代码中声明的 Numpy 数组有 11 个元素，每个元素作为一个特征进行 reshape(-1, 1)转换后，X 变成有 11 个一维元素的二维数组：

```
>>> print(observe_chain)
[[0]
 [2]
 [1]
 [1]
 [1]
 [1]
 [1]
 [1]
 [2]
 [2]
 [2]]
```

其中每个一维元素是一个可见层结点。如果整个 X 变量只由一条可见链组成，则在.predict()函数中可以不输入 lengths 参数，下面的.predict()语句与前者等效：

```
model.predict(X)
```

2. X 由 1 个可见链构成，每个可见结点有多个特征

用于可见结点由多个特征组成的情况，比如：

```
X= np.array([0,2,1,1,1,1,1,1,2,2,2,0]).reshape(-1,3)
model.predict(X, [4,])
```

X 数组中共有 12 个数值，详细查看 X 的情况：

```
>>> X
array([[0, 2, 1],
      [1, 1, 1],
      [1, 1, 2],
[2, 2, 0]])
```

其由 4×3 的二维数组组成，此时可见链的长度为 4。

提示： 对多特征的结点解码之前，对 model.startprob_、model.transmat_、model.emissionprob_等 HMM 模型参数的维度设置也要符合相应特征的数量。它们的维度设置同样也是通过 Numpy 数组完成的。

由于是 1 条可见链，本段代码中.predict()的 lengths 参数也可省略。

3. X 由多个可见链构成

可以在一条语句中输入多条可见链，这个特性由.predict()函数的 lengths 参数控制。比如：

```
X= np.array([0,2,1,1,1,1,1,1,2,2,2,]).reshape(-1,1)
model.predict(X, [4,7])
```

以上代码的 X 中每个可见结点由一个特征组成，共对两条可见链分别进行解码，第一条可见链长度为 4，第二条可见链长度为 7。

```
X= np.array([0,2,1,1,1,1,1,1,2,2,2,0]).reshape(-1,2)
model.predict(X, [2，1，3])
```

以上代码的 X 中每个可见结点由两个特征组成，共对 3 条可见链分别进行解码，第一条可见链长度为 2，第二条可见链长度为 1，第三条可见链长度为 3。

4. 查看解码返回

不管.predict()函数的输入由几条可见链构成，.predict()函数的输出都用一个一维数组返回所有隐藏结点解码，比如：

```
>>> observe_chain = np.array([0,2,1,1,1,1,1,1,2,2,2]).reshape(-1,1)
>>> print(model.predict(observe_chain))
[0 0 2 2 2 2 2 2 1 1 1]     # 返回数组的所有 11 个元素构成 1 个隐藏链
```

```
    print(model.predict(observe_chain, [5, 6]))
    [0 0 0 0 0 2 2 2 1 1 1]     # 返回数组的前 5 个元素是一个隐藏链，后 6 个元素是另一条
隐藏链
```

如果一次提交多条可见链进行解码，开发者需要自行对返回的隐藏链进行拆分。

6.2.5 EM 算法

EM 是最大期望算法的简称（Expectation–Maximization），它是一种迭代改进的算法。机器学习领域的很多算法其实是 EM 的一种具体化，包括第 4 章中的 K-means 聚类算法和后面将要学习的 Baum-Welch 等。这里先讨论 EM 算法，然后引入 Baum-Welch 过程。

1. EM 算法过程

EM 算法通过不断地迭代来找出模型最佳参数θ，原理如下。

◎ 定义系统评价方法（评价结果在有些论文/书籍中称为隐含变量）。

◎ 随机初始化模型参数θ。

◎ 不断重复迭代如下两步。

  i. E（expectation）步：用当前参数θ评价所有输入数据。

  ii. M（maximization）步：用评价结果重新计算θ参数，使得新的θ对输入数据获得更好的评价。

◎ 当 M 步对θ的改进不再显著时停止上述迭代，认为系统已经找到了最佳参数θ。

注意：这里的参数θ可以不只是单一参数，大多数的场景中是指一组变量$\theta1$、$\theta2$、$\theta3$……

下面用一个生活中的场景解释以上步骤。假设有一堆沙子，如何用一个天平将这堆沙子分为相等的两堆？读者可能马上想到一个重复尝试方法，这正是对EM算法的一个实践，将这个分沙子的过程对应到如下 EM 算法的步骤中。

◎ 定义评价方法：天平杠杆的水平角度越小，两堆沙子越接近相等。

◎ 初始化模型参数：先随意地将沙子分为两堆。

◎ 不断重复迭代。

i．E 步：查看天平杠杆的角度，发现某堆较重。

ii．M 步：从较重的一堆移出一些到较轻的一堆，使得天平杠杆角度比原来小。

◎　重复上述步骤，直到天平角度趋于水平不能更小为止。

2. EM 算法局限性

EM 算法与梯度下降有类似的局限性：该算法并不保证能找到全局最优参数，只能保证找到局部最优解。

以一个一维特征找最大值为例，如图 6-8 所示。如果 EM 算法的随机初始值在图 6-8 的左半边，则 EM 算法极有可能攀登到左侧的局部最高点；只有当随机初始值在图的右半边时才能攀登到全局最优点。

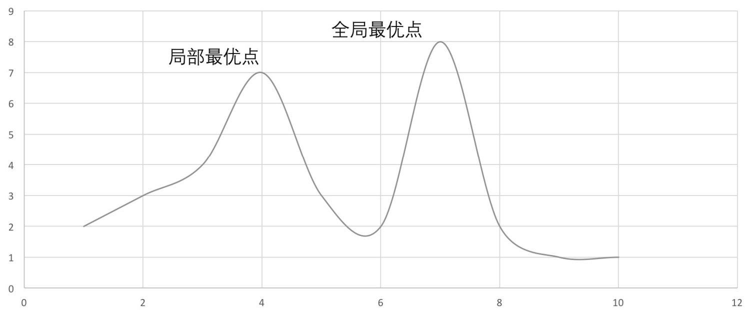


图 6-8　局部最优与全局最优

但是这个特点并没有影响 EM 算法的广泛应用，其原因有二：

◎　EM 算法是对复杂问题在全局最优解与时间复杂度之间的一个权衡，寻找全局最优解的算法在高维特征情况下通常超出了多项式级别的时间复杂度。

◎　在高维环境中，算法陷入局部最优解的概率较小。

因此，EM 算法是一种非常有“性价比”的算法。

6.2.6　Baum-Welch 算法

HMM 前两类问题都假定 HMM 参数已经给出，但是模型参数本身从何而来呢？这个问题主要有两种思路进行解决。

◎ 由足够多已有的隐藏层状态链表生成转换概率矩阵；由足够多的“隐藏状态<->可见状态”链表生成输出概率矩阵。

◎ 给出足够多的可见状态链表，根据其猜测输出概率和转换概率矩阵。

第一种方式本质上是一种手工整理海量数据的方法，需要对大量的数据进行人工标记，这在很多领域是不可能或成本过高的。

第二种方式就是本小节要介绍的 Baum-Welch 方法，Baum 正是隐马尔可夫模型的提出者。由于这种方式对隐藏层一无所知，通常不同的初始参数会导致不同的结果，所以在训练之后仍然需要对模型概率数据进行人工调研。但总体来说，这种方法相对第一种方法仍然更加经济高效。

1. 总体流程

Baum-Welch 采用的就是在 6.2.5 中介绍的 EM 算法，只不过在其出现的 20 世纪 60 年代，EM 算法还没有正式被提出，所以在 HMM 领域其只是被称为 Baum-Welch 算法。如下步骤将 Baum-Welch 对应到 EM 过程中。

◎ 定义评估方法，最大化可见状态链的前向概率α和后向概率β。

◎ 初始化模型参数：随机定义一套初始概率π、转换概率矩阵 $\boldsymbol{A}$、输出概率矩阵 $\boldsymbol{B}$。

◎ 不断重复迭代。

i．E 步：用当前的π、$\boldsymbol{A}$、$\boldsymbol{B}$ 计算出α和β（即前向-后向算法）。

ii．M 步：用α和β更新π、$\boldsymbol{A}$、$\boldsymbol{B}$。

◎ 重复上述步骤，当π、$\boldsymbol{A}$、$\boldsymbol{B}$ 的更新趋于静止时认为已经找到了模型最佳参数，停止迭代。

提示：此流程中的α和β就是在 6.2.1 中学习的 Forward-Backward Algorithm 中用到的前向概率和后向概率。

2. 扩展阅读：M 步细节

现在可能读者唯一会疑惑的就是“M 步：用α和β更新π、$\boldsymbol{A}$、$\boldsymbol{B}$”中，到底是如何用α和β计算出新的 HMM 参数的。这一部分需要比较扎实的贝叶斯理论公式基础，这里给出

简要细节，有兴趣的读者可以结合贝叶斯公式仔细思考其意义。

◎　计算中间变量γ，其含义是对于给定 Y 和 θ 在时间点 t 下隐藏层状态为 i 的概率：

$$\gamma_i(t)=P(X_t=i\,|\,Y,\theta)=\frac{P(X_t=i,Y\,|\,\theta)}{P(Y\,|\,\theta)}=\frac{\alpha_i(t)\beta_i(t)}{\sum_{j=1}^{N}\alpha_j(t)\beta_j(t)}$$

◎　计算中间变量 ξ，其含义是在两个连续时间点 t、t+1，它们的隐藏状态分别是 i、j 的概率。

$$\xi_{ij}(t)=P(X_{t+1}=i,X_{t+1}=i\,|\,Y,\theta)=\frac{P(X_t=i,X_{t+1}=j,Y\,|\,\theta)}{P(Y\,|\,\theta)}$$
$$=\frac{\alpha_i(t)a_{ij}\beta_j(t+1)b_j(y_{t+1})}{\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i(t)a_{ij}\beta_j(t+1)b_j(y_{t+1})}$$

◎　用 γ 和 ξ 更新 π、$\boldsymbol{A}$、$\boldsymbol{B}$：

$$\pi_i=\gamma_i(1)\text{、}\quad a_{ij}=\frac{\sum_{j=1}^{T-1}\xi_{ij}(t)}{\sum_{j=1}^{T-1}\gamma_i(t)}\text{、}\quad b_{ij}=\frac{\sum_{j=1}^{T-1}1_{if\ y_t=j}\gamma_i(t)}{\sum_{t=1}^{T}\gamma_i(t)}\text{。}$$

在如上公式中，Y 代表可见层链，T 代表时间序列，θ 是 HMM 参数的总称（包括 π、$\boldsymbol{A}$、$\boldsymbol{B}$），a、b、t 是 $\boldsymbol{A}$、$\boldsymbol{B}$、$\boldsymbol{T}$ 中的元素。

6.2.7　用 hmmLearn 训练数据

hmmLearn 中训练数据的方法沿用了 scikit-learn 中的命名习惯，即.fit()，其函数原型如下。

```
    fit(X, lengths=None)
    Estimate model parameters.
    An initialization step is performed before entering the EM algorithm. If you
want to avoid this step for a subset of the parameters, pass proper init_params
keyword argument to estimator’s constructor.
    Parameters:
    X : array-like, shape (n_samples, n_features)
        Feature matrix of individual samples.
    lengths : array-like of integers, shape (n_sequences, )
        Lengths of the individual sequences in X. The sum of these should be
```

```
n_samples.
    Returns:
    self : object
       Returns self.
```

所有三个 hmmLearn 中定义的 HMM 模型都用该方法进行数据训练，其内部算法就是前一节所学的 Baum-Welch 算法。

函数的参数 *X* 和 lengths 的含义与.predict()中的参数一致，使用方法也相同。可以通过这三个参数控制：

◎ 可见链结点的特征维数。

◎ 可见链的数量。

◎ 每条可见链的长度。

具体方法可参考.predict()一节，此处不再赘述。

6.3 连续型概率分布

截至目前本章一直用离散可见层状态输出概率模型讲解建模及算法原理，这种模型使用的是多项分布，即 hmmLearn 中的 MultinomialNB，它相对于连续型可见层状态分布的特点就是输出概率可以通过输出概率矩阵直接表示。

在众多场景中表现层状态需要建模成连续型概率分布，如金融分析、语音识别等领域。这种情况下由于可见状态无法穷举，所以无法以定长定宽的矩阵形式表达输出概率；同时由于连续型状态的值之间有大小关系，使其能够用正态分布表征，所以在 hmmLearn 中使用高斯分布和混合高斯分布来对其建模。

在 hmmLearn 中连续型模型在解决 HMM 三大问题的调用方式上与离散模型 MultinomialNB 保持一致，分别用.score()、.predict()、.fit()函数进行概率估计、隐藏层猜测、无监督训练。对它们的调用方式本节不再重述，而将重点放在分析连续模型与离散模型不同的模型参数上。本节先介绍多元高斯分布的基础知识，再介绍 hmmLearn 中两种高斯模型的具体使用方法。

6.3.1　多元高斯分布

由于 HMM 中每个可见层结点的状态可以由多个特征一起构成，其中每个特征都用高斯分布拟合，所以对可见层结点整体来说其服从的是多元高斯分布。

多元高斯分布（Multivariate Gaussian Distribution）是高斯分布在多特征情况下的扩展。

提示：本小节的目的是使读者理解 GaussianHMM 和 GMMHMM 中两个重要参数（均值矩阵和协方差矩阵）的原理。读者也可以先跳过本小节，当在下文中遇到这两个参数时再来回顾。

1. 普通高斯分布

回顾高斯分布的公式：

$$f(x \mid \mu,\sigma^2)=\frac{1}{\sqrt{2\pi\sigma^2}}\mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

其中有如下两个方程参数。

◎　μ：即均值（mean）、高斯分布期望值，即概率分布中心点。

◎　σ^2：即方差（variance）、描述分布的聚合程度，方差越小概率分布越集中。

注意：σ 本身被称为标准差（Standard Deviation），在 hmmLearn 中只用到方差，需注意区分。

2. 多元高斯分布

多元高斯分布与普通高斯分布的不同在于如下三点：

◎　输入参数由标量 x 变为向量 $\langle x_1,x_2,x_3,x_4,x_5,\ldots\rangle$。

◎　均值由标量 μ 变为向量 $\langle \mu_1,\mu_2,\mu_3,\mu_4,\mu_5,\ldots\rangle$，分别表示对应输入参数 $\langle x_1,x_2,x_3,x_4,x_5,\ldots\rangle$ 的均值。

◎　方差由标量 σ^2 变为 $N\times N$ 的协方差矩阵（Covariance Matrix），其中 N 是输入向量的维数。

如图 6-9 所示是一个引自 Wiki 的二维多元高斯分布的示例，其 X 与 Y 轴各是一个特征分量，分别服从高斯分布，结合在一起就是一个多元高斯分布。

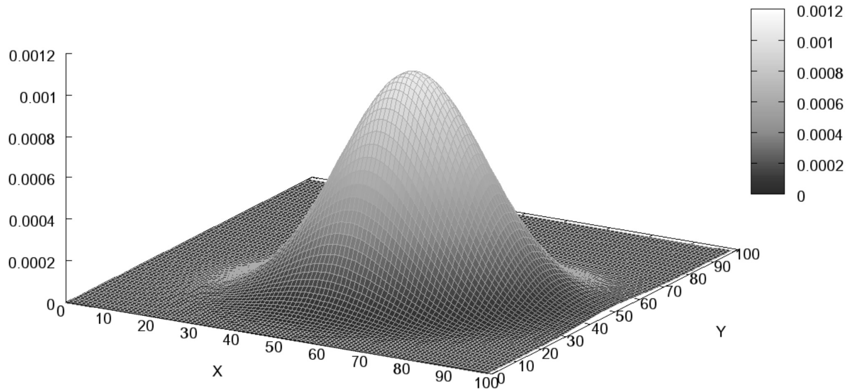


图 6-9　二维多元高斯分布的示例（取自 Wiki）

3. 协方差矩阵

多元高斯分布中唯一比较难理解的是方程参数协方差矩阵，理解协方差矩阵要从理解协方差开始。协方差的定义为：$\mathrm{cov}(X,Y)=E[(X-\mu_X)(Y-\mu_Y)]$。

其中 $E[]$是期望函数，协方差的意义是用来衡量两个特征之间的相关性，有如下特点：

◎　协方差为正时两个特征正相关，即 X 变大时 Y 也变大，X 变小时 Y 也变小。

◎　协方差为负时两个特征负相关，即 X 变大时 Y 变小，X 变小时 Y 变大。

◎　协方差绝对值越大，两个特征相关性越大。

协方差仅能表现两个特征之间的相关性，如何表征多个特征的相关性呢？答案就是协方差矩阵，其形式如图 6-10 所示。

$$\boldsymbol{\Sigma}=\begin{bmatrix} E[(X_1-\mu_1)(X_1-\mu_1)] & E[(X_1-\mu_1)(X_2-\mu_2)] & \cdots & E[(X_1-\mu_1)(X_n-\mu_n)] \\ E[(X_2-\mu_2)(X_1-\mu_1)] & E[(X_2-\mu_2)(X_2-\mu_2)] & \cdots & E[(X_2-\mu_2)(X_n-\mu_n)] \\ \cdots & \cdots & \ddots & \cdots \\ E[(X_n-\mu_n)(X_1-\mu_1)] & E[(X_n-\mu_n)(X_2-\mu_2)] & \cdots & E[(X_n-\mu_n)(X_n-\mu_n)] \end{bmatrix}$$

图 6-10　协方差矩阵

从图 6-10 可知，协方差矩阵必定是一个方阵，其元素就是横纵坐标对应特征之间的

协方差。有了协方差矩阵，就可以结合均值向量计算多特征结点的整体概率分布了。具体计算过程涉及行列式等概念，此处不再展开，有兴趣的读者可以参考概率论书籍，比如陈希儒教授的《概率论与数理统计》。

6.3.2　GaussianHMM

GaussianHMM 是两种高斯分布中较简单的形式，它假定：

◎　结点中的每个维度都符合高斯分布，因此输出概率是一个多元高斯分布。

◎　隐藏层状态是对每个可见层特征分别识别出的均值与方差。

◎　状态维度之间可以有关联关系，通过输出概率协方差矩阵来体现。

GaussianHM 的构造方式如下：

```
class hmmlearn.hmm.GaussianHMM
(n_components=1, covariance_type='diag', min_covar=0.001,
 startprob_prior=1.0, transmat_prior=1.0, means_prior=0, means_weight=0,
 covars_prior=0.01, covars_weight=1, algorithm='viterbi',
 random_state=None, n_iter=10, tol=0.01, verbose=False, params='stmc',
 init_params='stmc')
```

下面逐个介绍其参数。

◎　n_components：隐藏层结点的状态数量。

◎　covariance_type：输出概率的协方差矩阵类型，可选值如下。

　　i．spherical：球面协方差矩阵，即矩阵对角线上的元素都相等，且其他元素为零。

　　ii．diag：对角协方差矩阵，即对角线元素可以是任意值，其他元素为零。

　　iii．full：完全矩阵，即任意元素可以是任意值。

其中 spherical 倾向于使所有特征使用相同的方差；而 full 使用各自不同的方差，但同时需要更多的训练数据；diag 是一种折中的方法，也是参数默认值。

◎　min_covar：协方差矩阵中对角线上的最小数值，该值设置得越小模型对数据的拟合就越好，但更容易出现过度拟合。

◎　startprob_prior：第一个隐藏结点的初始概率矩阵。

◎ transmat_prior：转换概率矩阵。

◎ means_prior，means_weight：先验隐藏层均值矩阵。

◎ covars_prior、covars_weight：先验隐藏层协方差矩阵。

◎ algorithm：解码算法，可选值为 viterbi 或 map。其中 viterbi 算法已经介绍过，map 比 viterbi 更快速但非全局最优解。

◎ random_state：随机种子，用于在 Baum-Welch 算法中初始化模型参数。

◎ n_iter：Baum-Welch 算法最大迭代次数。该值越大，训练模型对数据的拟合度越高，但训练耗时越长。

◎ tol：Baum-Welch 算法停止迭代的容忍阈值。该值越小（必须>=0），训练模型对数据的拟合度越高，但训练耗时越长。

◎ verbose：是否打印 Baum-Welch 每次迭代的调试信息。

◎ params：在训练过程中更新哪些 HMM 参数。可以是四个字母中的任意几个组成的字符串：“s”“t”“m”“c”。其中 *s* 指代初始概率，*t* 是转换概率，*m* 是高斯均值矩阵，*c* 是高斯协方差矩阵。

◎ init_params：在训练开始之前使用哪些已有概率矩阵初始化模型。与 params 参数一样可以包括四个字母的任意组合：“s”“t”“m”“c”。

6.3.3 GMMHMM

GMMHMM 适合用于隐藏层状态是连续类型且假设输出概率符合 GMM 分布（Gaussian mixture model，混合高斯分布）的情况。它与 GaussionHMM 的不同点仅在于它假设输出概率使用 GMM 而非简单的 Gaussian 分布，在 hmmLearn 中原型如下：

```
class hmmlearn.hmm.GMMHMM
(n_components=1, n_mix=1, min_covar=0.001, startprob_prior=1.0,
 transmat_prior=1.0, weights_prior=1.0, means_prior=0.0, means_weight=0.0,
 covars_prior=None, covars_weight=None, algorithm='viterbi',
 covariance_type='diag', random_state=None, n_iter=10, tol=0.01,
 verbose=False, params='stmcw', init_params='stmcw')
```

其中大部分参数与 GaussianHMM 中的含义一样，不再重复说明，不同点有如下两个。

◎ n_mix：混合高斯概率分布中高斯分布的数量，如果 n_min 等于 1，则 GMMHMM 退化为 GaussianHMM。

◎ means_prior，means_weight，covars_prior，covars_weight：应该注意这四个参数虽然名称与 GaussianHMM 一致，但其维数随着 n_mix 的设置而不同。

如图 6-11 所示取自 Wiki 网站，形象地解释了混合高斯模型的概率分布。

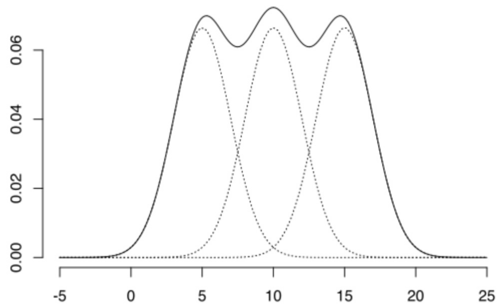


图 6-11　混合高斯模型举例（取自 Wiki）

图中的混合高斯模型由三个高斯分量（n_mix=3）组成。实线为混合高斯模型的总体密度曲线，虚线为三个高斯分量的密度函数。

由于理论上任何密度函数都可以由混合高斯模型拟合，所以 GMMHMM 有非常大的适用场景。

6.4 实战：股票预测模型

用 hmmLearn 官网上的经典 GaussianHMM 开发实战案例“股票走势预测”来结束本章。本案例从雅虎金融网站获取真实的历史交易数据，在本地训练并分析历史走势，尝试寻找出历史走势规律以预测未来。

6.4.1 数据模型

雅虎金融网站是很多业余金融爱好者的实盘数据来源，网上的许多程序直接从 yahoo.com 下载后分析数据。但由于雅虎金融 WebService 接口格式经常变化，一些实时功能时有时无，因此这样的在线架构无法保证程序能够一直正常运行。

1. 数据格式

为保证读者能够顺利实践，本书代码使用事先下载好的数据，CSV 文件保存在代码目录中。目前原始数据文件格式如图 6-12 所示。

	A	B	C	D	E	F	G
1	Date	Open	High	Low	Close	Adj Close	Volume
2	21/09/2012	9.85	10.09	9.8	9.82	9.528357	66200
3	24/09/2012	9.83	10.07	9.61	9.9	9.60598	44500
4	25/09/2012	9.93	9.99	9.57	9.58	9.295483	54400
5	26/09/2012	9.63	9.72	9.55	9.59	9.305186	29100
6	27/09/2012	9.6	9.71	9.44	9.57	9.28578	55300
7	28/09/2012	9.55	9.89	9.49	9.67	9.382811	26600
8	01/10/2012	9.73	9.88	9.32	9.55	9.266374	34000
9	02/10/2012	9.61	9.76	9.44	9.52	9.237267	19000
10	03/10/2012	9.5	9.51	9.26	9.28	9.004395	83700
11	04/10/2012	9.36	9.73	9.36	9.64	9.353702	46300
12	05/10/2012	9.69	9.73	9.52	9.54	9.256671	23700
13	08/10/2012	9.54	9.54	9.34	9.41	9.130532	80100

图 6-12　原始数据文件格式

从图中可知，原始数据是以日为单位的某股票基础交易数据，其中的数据包括日期、开盘价、当日最高价、当日最低价、收盘价、调整收盘、当日交易量。

2. 数据更新

希望更新最新数据的读者可以到雅虎网站重新下载，目前可用的下载地址是：https://finance.yahoo.com/quote/CSV/history?p=CSV，下载界面如图 6-13 所示。

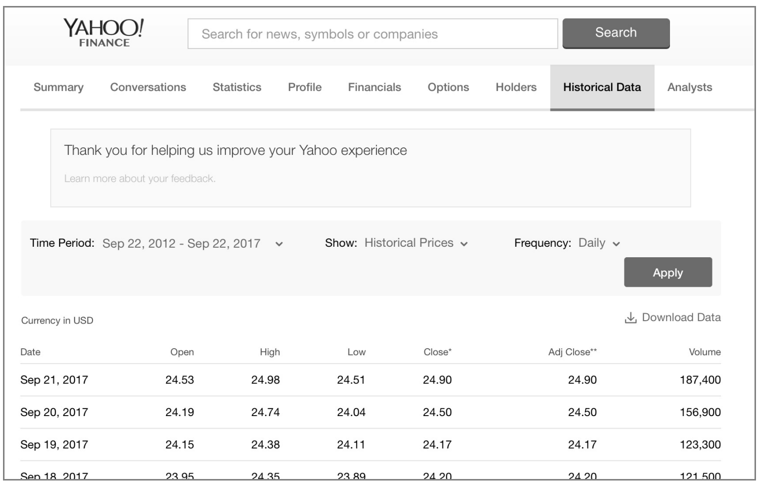


图 6-13　从雅虎网站下载最新数据

在图 6-13 的 Time Period 中选择下载历史时间段，单击“Apply”按钮查询新数据后再单击“Download Data”链接即可下载最新数据。

6.4.2　目标

既然是做股票数据分析，最直接的目标就是用历史数据进行建模，并通过模型对新数据进行分析，以得出未来是涨是跌的预测。下面用 HMM 模型通过如下几点描述这个目标。

◎ HMM 时间轴：由于数据模型是日交易信息，所以本模型的时间轴以日为单位，即每一天是一个 HMM 状态结点。

◎ 可见层特征：选取数据文件中的两个重要数据作为可见层特征，即收盘涨跌值、交易量。因为这些属性为连续值，所以选用 Gaussian 模型。

注意：收盘涨跌值特征并未在数据模型中直接体现，它要通过计算前后两天收盘价的差值获得。

◎ 隐藏层状态：定义三种状态，对应于涨、平、跌。

◎ 预测方式：通过查看隐藏层转换矩阵的转换概率，根据最后一个结点的隐藏状态预测未来一天的涨跌可能。

以上几点中理解最后一条尤为重要，因为隐藏层的涨跌状态只受当天及之前表示层特征的影响，所以其本身并不能决定下一天的走势。而要预测下一天的隐藏层状态，需要结合转换概率矩阵进行分析。

6.4.3　训练模型

明确了目标后就可以开始写代码了，首先引入需要用到的 Python 包：

```
import datetime
import numpy as np                              # 用 numpy 进行特征运算
from matplotlib import cm, pyplot as plt   # 用 matplot 绘图
from matplotlib.dates import DayLocator
from hmmlearn.hmm import GaussianHMM        # 选择的 HMM 模型
```

从数据文件 sample_stock.csv 读取特征数据。

```
def load_sample_stock():
```

```
        import csv
        ret = []
        with open('hmm/sample_stock.csv', newline='') as csvfile:
            reader = csv.DictReader(csvfile)
            for idx, row in enumerate(reader):
                ret.append((datetime.datetime.strptime(row["Date"],
"%d/%m/%Y"), float(row["Close"]), float(row["Volume"])))
        return ret

    quotes = load_sample_stock()
```

该函数用 CSV 库以只读方式读取 CSV 文件，然后逐行读取需要的特征到一个列表中，再用如下代码将特征转换成 Numpy 数组：

```
    dates = np.array([q[0] for q in quotes], dtype=datetime.datetime)[1:0]
    close_v = np.array([q[1] for q in quotes])
    volume = np.array([q[2] for q in quotes])[1:]
```

注意其中日期 dates 和交易量 volume 都去除了读到的第一天数据，这是因为第一天会被用来计算第二天的涨跌值特征，所以马尔可夫链实际是从第二天开始训练的。计算涨跌值特征只需要一行代码即可完成：

```
    diff = np.diff(close_v)
```

其中 np.diff()是 Numpy 用于计算数组中前后两个值差额的函数。

通过如下代码将涨跌额、交易量合并为一个二维数组，作为后续 HMM 模型的可见层链输入：

```
    X = np.column_stack([diff, volume])
```

现在就可以初始化一个 GaussianHMM 模型，并用 X_train 变量进行模型训练了：

```
    model = GaussianHMM(n_components=3, covariance_type="diag", n_iter=1000)
    model.fit(X)
```

在 GaussianHMM 初始化参数中，n_components 参数指定了使用 3 个隐藏层状态；covariance_type 定义了协方差矩阵类型为对角线类型，即每个特征的高斯分布有自己的方差参数，相互之间没有影响；n_iter 参数定义了 Baum-Welch 的最大迭代次数。

在 fit()函数完成训练后，model 已经成为了拟合该股票涨跌情况的 HMM 模型。

6.4.4 分析模型参数

本节即解读股票 HMM 的训练结果含义，也帮助读者学习连续型可见层概率分布的模型参数。

1. 均值矩阵

由于使用的是 Baum-Welch 无监督训练法，即在训练中完全没有给出隐含状态的任何信息，所以需要在训练后观察均值矩阵的具体数值才能知道每个隐含状态的含义。

通过如下命令打印均值矩阵：

```
print("Model means:")
print(model.means_)
```

得到如下数据：

```
[[ -6.38539802e-03   5.89138094e+04]
 [ -7.37591501e-02   4.25599160e+05]
 [  4.39627713e-02   1.42960906e+05]]
```

以上矩阵共有三行，其中每一行代表一个隐藏层结点状态。因为可见层输入采用了两个输入特征（涨跌幅、成交量），所以每个结点有两个元素，分别代表该状态的涨跌幅均值、成交量均值。

由于系统的目标预测涨跌幅，所以这里只关心第一个特征的均值，有如下结论：

◎ 状态 0 的均值是–6.38539802e–03，约为–0.0064，认为该状态是“平”。

◎ 状态 1 的均值是–7.37591501e–02，约为–0.074（跌 7 分钱），得知该状态是“跌”。

◎ 状态 1 的均值是 4.39627713e–02，约为 0.044（涨 4 分钱），得知该状态是“涨”。

2. 协方差矩阵

通过如下命令打印协方差矩阵：

```
print("Covariance means:")
print(model.covars_)
```

得到数据如下。

```
[[[  4.85808647e-02   0.00000000e+00]
  [  0.00000000e+00   4.88397764e+08]]

 [[  1.11790242e+00   0.00000000e+00]
  [  0.00000000e+00   1.11031189e+11]]

 [[  1.47535241e-01   0.00000000e+00]
  [  0.00000000e+00   3.28682612e+09]]]
```

其中三个协方差矩阵，分别对应隐藏状态“平”“跌”“涨”。另外，每个协方差矩阵的非对角线元素都为 0，这对应了对 GaussianHMM 初始化参数 covariance_type == "diag" 的配置。

与均值矩阵一样，这里只关心涨跌幅特征的方差（每个协方差矩阵的左上角数值），而不去管成交量特征的方差（右下角数值）。可以得到的信息为：

◎ 状态“平”的方差为 4.85808647e–02，是三个状态中方差最小的，也就是说该状态的预测非常可信。

◎ 状态“跌”的方差为 1.11790242e+00，是三个状态中方差最大的，即该状态的变化范围较大，并非很可信。

◎ 状态“涨”的可信度居中。

有了这些均值与方差的数值，利用正态分布表甚至可以估计涨跌幅的可信区间。

3. 转换概率矩阵

可以用如下命令打印出训练后 HMM 的转换概率矩阵：

```
print("Transition matrix:")
print(model.transmat_)
```

得到如下形式的矩阵：

```
 [[ 0.82016839  0.01737474  0.16245687]
 [ 0.03813588  0.230168    0.73169612]
 [ 0.24423382  0.04391393  0.71185225]]
```

矩阵的三行仍旧代表三个隐含状态，其含义如下。

◎ 第一行最大的数值是 0.82016839 ，因此“平”倾向于保持自己的状态，即第二

天仍旧为“平”。

◎ 第二行最大的数值是 0.73169612，得知“跌”后第二天倾向于变为“涨”，但同时也有不小的概率仍旧是“跌”（0.230168），而“跌”后变为“平”的可能性则非常小。

◎ 根据第三行状态，“涨”后第二天仍旧倾向于“涨”，而不太可能马上变“跌”。

根据以上转换概率，可以看出该股票的长线态势是非常看好的。

6.4.5　可视化短线预测

现在可以开始编写程序预测最后一天的涨跌可能了。

1. 可视化预测

首先选取最近的一段数据调用.predict()方法预测该段数据的隐藏状态：

```
X = X[-26:]
dates = dates[-26:]
close_v = close_v[-26:]
hidden_states = model.predict(X)
```

以上代码已经完成了隐藏层的状态猜测，接下来可视化之前的分析结果。如下代码生成列表变量 means_和 vars_保存每种隐藏状态的均值和方差：

```
means_ = []                                    # 每种状态的均值
vars_ = []                                     # 每种状态的方差
for i in range(model.n_components):
   means_.append(model.means_[i][0])
vars_.append(np.diag(model.covars_[i])[0])
```

利用转换矩阵，计算当前状态后一天的预测均值：

```
predict_means = []                                # 每种状态后一天的最可能均值
for idx, hid in enumerate(range(model.n_components)):
   comp = np.argmax(model.transmat_[idx])         # 最可能的第二天状态
   predict_means.append(means_[comp])
```

如下代码使用 matplot 分别画出这段日期的状态图，对于不同的预测状态使用不同的端点形状和颜色进行示意。

```
fig, axs = plt.subplots(model.n_components+1, sharex=True, sharey=True)
for i, ax in enumerate(axs[:-1]):
    ax.set_title("{0}th hidden state".format(i))

    mask = hidden_states == i
    yesterday_mask =  np.concatenate(([False], mask[:-1]))
    if len(dates[mask])<=0:
        continue
if predict_means[i] > 0.01:
    # 上涨预测，端点形状是上三角，用红色
    ax.plot_date(dates[mask], close_v[mask], "^", c="#FF0000")
elif predict_means[i] < -0.01:
    # 下跌预测，端点形状是下三角，用绿色
    ax.plot_date(dates[mask], close_v[mask], "v", c="#00FF00")
else:
    # 平，端点形状是加号，用黑色
    ax.plot_date(dates[mask], close_v[mask], "+", c="#000000")

# Format the ticks
ax.xaxis.set_minor_locator(DayLocator())

ax.grid(True)
ax.legend(["Mean: %0.3f\nvar:   %0.3f"%(predict_means[i],
        predict_vars[i])], loc='center left', bbox_to_anchor=(1, 0.5))
```

最后用一个子图打印出这段日期的真实走势用作对比：

```
axs[-1].plot_date(dates, close_v, "-", c='#000000')
axs[-1].grid(True)
box = axs[-1].get_position()
axs[-1].set_position([box.x0, box.y0, box.width * 0.8, box.height])
ax.xaxis.set_minor_locator(DayLocator())
```

调整格式并显示作图：

```
fig.autofmt_xdate()
plt.subplots_adjust(left=None, bottom=None, right=0.75, top=None,
                wspace=None, hspace=0.43)
plt.show()
```

最后的效果如图 6-14 所示。

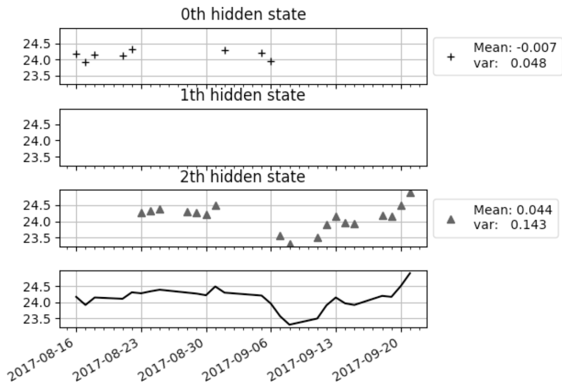


图 6-14　涨跌状态最后的效果

由图 6-14 可知，这个月中大部分隐藏状态是“平”或者“涨”，其中“涨”的数量更多，由最下方真实走势图可见当月结果确实为上涨。最后一天的结点落在了“2th hidden state”中，其预测状态为上涨（均值为 0.044，方差为 0.143）。

2. 调整状态数量

以上已经完成了 HMM 股票分析，接下来可以尝试调整一些训练参数，再观察结果，如图 6-15 所示是当 n_components 也就是隐藏状态数量为 10 时的结果。

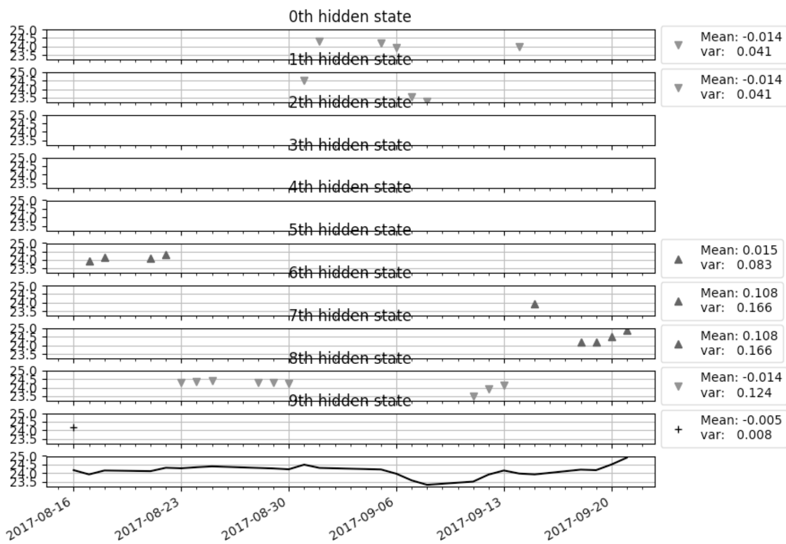


图 6-15　隐藏状态数量为 10 时的结果

其最后一天的预测结果仍为上涨，但更好地找到了中间的一些下跌状态。

说明：本节的完整代码保存在 hmm/plot_hmm_stock_analysis.py 文件中，可以运行该文件实践上述内容。

6.5 本章内容回顾

◎ 学习了隐马尔可夫模型的适用场景和建模方法。

◎ 深入理解解决隐马尔可夫模型三大问题的算法：前后向算法、Viterbi 算法、Baum-Welch 算法。

◎ hmmLearn 中三大模型 MultinomialHM、GaussianHMM，GMMHMM 的应用。

◎ 案例分析：用 GaussianHMM 进行股票走势分析。

7

第 7 章 贝叶斯网络

贝叶斯网络是一种基于概率模型的综合工具，它可以将客观数据与领域专家的先验知识相结合，对具有多重依赖关系的若干事件组成的复杂场景进行建模，并应用于后续推理。本章介绍贝叶斯网络推理和构建的基本原理，并用 PyMC3 实践相关开发，主要内容如下。

◎ 典型贝叶斯问题：通过有趣的实例了解贝叶斯网络的应用场景。

◎ 网络静态结构：贝叶斯网络中点、线结构的作用。

◎ 概率运算：联合概率、边际概率、条件概率的概念与计算。

◎ VE 推理：使用贝叶斯网络进行概率推理的基本算法。

◎ 网络构建：讨论贝叶斯网络构建的几种方法。

◎ 近似推理：介绍马尔可夫链蒙特卡洛（MCMC）、变分贝叶斯（VB）两种近似推理算法原理。

◎ 案例：基于 PyMC3 开发胸科疾病诊断网络。

7.1 什么是贝叶斯网络

贝叶斯网络（Bayesian Network）是一种用点表示事件条件概率、用边表示事件依赖关系的有向无环图（Directed Acyclic Graph，DAG），实际上该网络表达了场景内所有事件的联合概率分布，因此可以完成后续任何基于条件概率、边际概率的推理应用。因此，贝叶斯网络也被称为信念网（Belief Network）。不知是否出于刻意，这两种表达同一模型的模型名称英文简写都是 BN。

7.1.1 典型贝叶斯问题

概率是对不确定事件的定量表达方式，贝叶斯公式是基本的概率规律。虽然它们是有些让人敬畏的数学概念，但人们生活中常会无意地表达概率想法。比如当同学小林说“这天可能要下雨了”，也许他的意思是“下雨的概率超过 70%”；当他说“下雨了也许女朋友会打电话给我”，那大概是这个意思“下雨的时候，女朋友有 30%的可能给我打电话”；而当他说“几乎每次打电话女朋友都问我要礼物”，这相当于“女朋友打电话时问我要礼物的概率达到 90%”。

一般来说定性的表达方式在生活中是足够的，如果有人常使用后面这种数字的表达方式，即使没有让人产生高高在上的距离感也多少会感觉有些故作姿态。但是当面对逻辑推理问题时，定性的表达方式就显得很不够了。假设有朋友问小林：“今晚女朋友会问你要礼物吗？”如果这是需要慎重回答的问题（朋友用这个问题与他打赌了一个月的早餐），那就最好用概率论推理来回答这个问题了。

本书之前已经学习了多种概率模型，读者可能马上会意识到这个朋友问的问题一点也不简单。要回答这个问题，天气和运气都是需要考虑的。此外，前面给出的三个已知条件根本不够用，最好还能知道天气不错的情况下女朋友打电话来的概率如何，女朋友晚上与小林见面并且要礼物的概率如何等。现实中如果真的想象这个问题可能会发现多得数不清的影响因素，比如第二天学校是否有考试、是否有女朋友喜欢看的电影、通过微信索要礼物等。此处进行适当简化，假设只考虑四种最重要的事件，小林的脑海中可能会出现如图 7-1 所示的思路图。

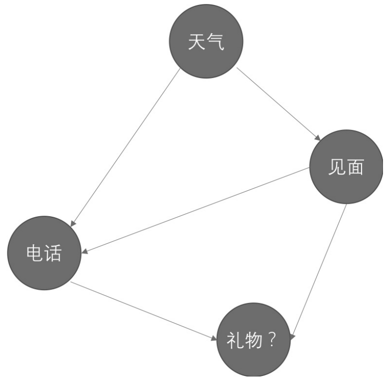


图 7-1　思路图

在图 7-1 中列出了会影响赌注问题的四个事件，而且它们并非独立，天气会影响小林与女朋友当晚是否会有电话、是否会见面；当晚见面与否会影响当晚是否会有电话的概率；女朋友在见面或者电话中都有可能问小林要礼物；是否索要礼物不与天气产生直接关系，但实际上通过另两个结点有间接关系。

有了这个思路图只是第一步，接下来再各个击破地仔细考虑每种情况发生的具体概率，最终才能推导出“晚上女朋友是否问小林要礼物”这个事件发生的可能性。

实际上，这整个过程就是一个贝叶斯网络从构建→拟合→推理的过程。虽然图 7-1 的网络用基本的概率公式就可以解决（具体方法后续讨论），但因为贝叶斯网络可以是任意复杂的有向无环图，随着网络规模的增大在网络构建和推理方面衍生出了多种确定性的或近似性的算法。

7.1.2　静态结构

贝叶斯网络与前一章学习的隐马尔可夫模型（HMM）是独立发展的两种基于图论的概率建模手段。与 HMM 按时间轴排列的两层链状模型不同，贝叶斯网络可以是任意的有向图。图中的每个结点都是对系统中一个事件的概率描述，并且用箭头线表达非独立事件之间的依赖关系。

1. 结点 CPD

在图 7-1 中已经用箭头表示了事件之间的依赖关系，因此如果要将其进一步转化成贝叶斯网络，只需要为每个结点配置条件概率，如图 7-2 所示。

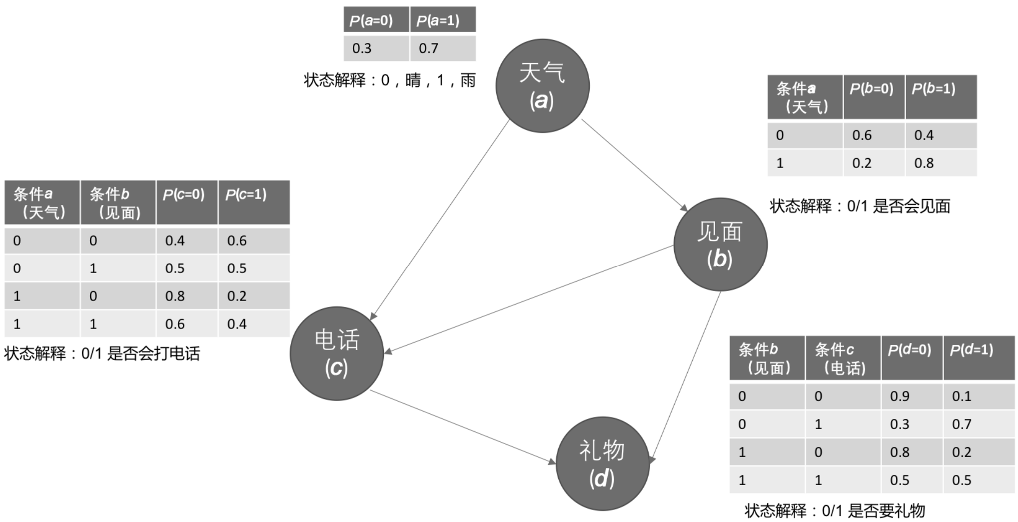


图 7-2　加入结点 CPD 的贝叶斯网络

在 BN 中描述概率的方式是每个结点上的条件概率分布（Conditional Probability Distributions，CPD），其中的“条件”就是每个结点的父结点。网络中没有父结点的结点是独立结点，它的 CPD 也就是该结点本身的概率分布。对于图 7-2：

◎ “天气”没有父结点，因此它只有一组概率分布，即 $P(a=0)=0.3$，$P(a=1)=0.7$。

◎ “见面”有一个父结点“天气”，意味着在不同的天气下有不同的见面可能性。因此需要确定在父结点所有状态下本结点各自的概率分布，也就是 $P((b=0\,|\,a=0))=0.6$，$P((b=0\,|\,a=1))=0.3\ldots$

◎ “电话”和“礼物”都各自有两个父结点，导致它们的 CPD 条目进一步增大。

2. BN 的实质是对联合概率的描述

至此，图 7-2 已经是一个完整的贝叶斯网络，它的实质是用 DAG 和 CPD 描述场景中所有事件的联合概率，即 $P(a,b,c,d)$，根据该联合概率可以计算出任何相关的条件概率和边际概率。

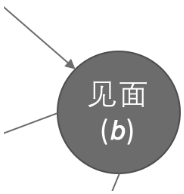
比如，如果有问题“小林与女朋友晚上见面的概率是什么？”如果此时晚上的天气是确定的，比如 a=1，则直接查见面事件的 CPD 即可得出结果 $P((b=1\,|\,a=1))=0.8$；而如果此时天气还未知，就需要考虑所有天气条件，即：

$$P(b) = \sum_a P(a)P(b \mid a) = 0.3 \times 0.4 + 0.7 \times 0.8 = 0.68$$

下一节将详细讨论各种概率的换算问题。

3. 多项式分布情况

图 7-2 描述的是一种极端简单的情况，其中每个结点的状态只有两种，即二项分布。而理论上贝叶斯网络中允许结点用任何分布描述 CPD，多项式分布就是另一种常见情况。多项式分布下每个结点的状态可以有多个值，比如仍考虑天气与见面两种事件，现假设将天气情况分为晴、阴、雨、雪四种情况，见面事件被分为不见面、学校见面、校外见面三种情况，则可能的见面事件的 CPD 如图 7-3 所示。



条件a（天气）	P(b=0)	P(b=1)	P(b=2)
0(晴)	0.6	0.2	0.2
1(阴)	0.2	0.3	0.5
2(雨)	0.1	0.5	0.4
3(雪)	0.3	0.7	0

图 7-3　多项式分布的 CPD

这样，b 结点的 CPD 由原来的 $2 \times 2 = 4$ 个概率值变为 $4 \times 3 = 12$ 个概率值，其中每行概率的和仍然保持为 1。而假设“电话”事件也分为三种状态，那么电话事件的 CPD 将由原来的 8 个概率值分裂成 $4 \times 3 \times 3 = 36$ 个概率值。可以想象，随着结点状态的复杂化计算 CPD 的过程也将越来越烦琐。

4. 连续变量的情况

在事件的状态是可枚举值的时候，无论如何都可以通过上述这样的二项分布、多项式分布等离散分布形式进行计算。但有时系统中会有用连续变量描述的结点，比如礼物金额，此时可以有两种策略处理。一种是将连续变量离散化后进行多项式等分布的建模；另一种就是不得不在 CPD 中引入连续分布，当然最常见的是高斯分布 $N(\mu, \sigma)$。

在用高斯分布对结点概率建模时的 CPD 如图 7-4 所示。



图 7-4　连续变量的 CPD

图 7-4 演示了两种连续概率建模方式，其一是当父结点是离散类型时，可以对父结点的每种条件都建立独立的高斯分布；另一种当父结点也是连续变量时，可以直接将父结点的值作为高斯分布的超参数。当然还有很多种其他的建模方式，需要领域专家结合专业知识确定 CPD 的具体形式。

7.1.3　联合/边缘/条件概率换算

现在已经知道了贝叶斯网络是对系统所有事件联合概率的表达，下面从概率基本原理的角度讲解通过 BN 进行推理的可行性。

1. 联合概率分布（Joint Probability Distribution）

从第 3 章开始本书已经多次接触了联合概率，在进行相关计算之前再次明确它的概念：联合概率是在多元变量描述的系统中“每个变量分别满足各自条件时”的概率。如果将联合概率看成由多元变量到概率值的映射，那么该函数就是联合概率分布函数，它的所有“自变量→因变量”对构成了联合概率分布。联合概率一般用 $P(a,b,c,\ldots,n)$ 这样的符号表示，或者在不引起歧义的情况下省略逗号成为 $P(abc\ldots n)$，其中 $a,b,c,\ldots,n$ 是系统中的所有变量。

2. 边缘概率分布（Marginal Probability Distribution）

设联合概率分布是函数 $f(a,b,c,d)$，用该函数查找概率值时需要给出系统中所有变量的值。但有时需要在某些变量未知的情况下查询概率，这样的函数就是边缘概率函数。边缘概率的表达方式与联合概率一样，其区别是通过边缘概率查询概率时只须给出系统中的部分变量。比如设系统中有变量 $a,b,c,\ldots,n$，那么任意的概率函数 $g(a,b)$，$g(b,d,g,n)$，$g(b)\ldots$ 都是该系统的边缘概率分布函数。

3. 条件概率分布（Conditional Probability Distribution）

条件概率用 $P(a,b|i=?,j=?)$ 这样的形式表达，表示在已知某些变量（本例 i、j）的情况下求另一些变量(a, b)给定值的概率。在条件概率分布函数 $f(a,b|i=?,j=?)$ 中，a、b 是函数变量，而 i，j 作为固定值是函数超参数。

4. 用联合分布求边缘分布

在图 7-2 中，类似“想知道被索要礼物的可能性”这类问题就是一个求边缘分布的问题。设已知联合概率分布函数 $f(x,y,z)$，想要求边缘分布 $g(y)$ 只需在联合分布上逐个对未知变量进行积分，即

$$g(y)=\iint_{-\infty}^{\infty} f(x,y,z)\mathrm{d}x\mathrm{d}z$$

对于离散变量，也就是

$$g(y)=\sum_{z}\sum_{x} f(x,y,z)$$

5. 用联合概率分布求条件概率分布

在图 7-2 中，类似“已知小林与女朋友有过电话，问天气如何”这样的问题是典型的条件概率问题。这类问题可直接套用公式

$$P(x|y=?)=\frac{P(x,y=?)}{P(y=?)}$$

进行计算，即将条件概率转换成边缘概率或联合概率。公式中 $y=?$ 是 y 的分布中对满足某种条件的筛选。

6. 用条件概率和边缘概率求联合概率

在概率论中，如果已知系统中的所有条件分布，是无法获得联合分布的。但如果同时已知条件概率与条件变量的边缘概率，则可以求得联合概率，即

$$P(x,y)=P(x|y)P(y)$$

读者应该可以发现，此公式只是求条件概率公式的简单变形。

7. 贝叶斯公式

将如上公式进行简单的组合，就可以得到“用条件概率求条件概率”的贝叶斯公式：

$$P(x \mid y)=\frac{P(x,y)}{P(y)}=\frac{P(y,x)}{P(y)}=\frac{P(y \mid x)P(x)}{P(y)}$$

这种情况适用于很容易求得 $P(y \mid x)$、$P(x)$、$P(y)$ 的时候。

7.1.4 链式法则与变量消元

贝叶斯网络中的链式法则（Chain Rule）是简化网络中联合概率计算的基本工具，而变量消元（Variable Elimination）则是通过联合概率进行推理的工具。

1. 链式法则

利用 7.1.3 节的各种换算关系，现在可以理解，为什么贝叶斯网络中明明只保存了所有结点的条件分布，却说整个网络能表达系统的联合分布了。

- ◎ 贝叶斯网络是一个有向无环图，所以至少会有一个没有父亲的“根”结点。
- ◎ 由于没有任何“条件”，则根结点的 CPD 本身就构成了自己的边缘分布。
- ◎ 结合根结点边缘分布与第一层子结点的 CPD 条件分布，就可以计算出根结点与子结点的联合分布，该联合分布也就构成下一层结点的边缘分布。
- ◎ 如此逐层向 DAG 的深处遍历，可以计算出所有结点的联合分布。

而这个步骤用概率公式表达，就是联合分布可通过所有结点的 CPD 连乘获得，图 7-2 中贝叶斯网络的联合概率就是：

$$P(a,b,c,d)=P(a)\cdot P(b \mid a)\cdot P(c \mid a,b)\cdot P(d \mid b,c)$$

这就是贝叶斯网络链式法则。

2. 变量消元

通过链式法则获得的联合概率公式能确切地计算出所有事件组合的概率，但现实中的问题往往只是求系统中的某些条件概率和边缘概率问题。比如，图 7-2 中的 BN 如何计算

$P(a|d=1)$，即已知小林的女朋友索要了礼物，求当天天气的概率分布，此时用上一小节中基本的概率转换运算可知：

$$P(a|d=1)=\frac{P(a,d=1)}{P(d=1)}=\frac{\sum_c\sum_b\left(P(a)\cdot P(b|a)\cdot P(c|a,b)\cdot P(d=1|b,c)\right)}{\sum_c\sum_b\sum_a\left(P(a)\cdot P(b|a)\cdot P(c|a,b)\cdot P(d=1|b,c)\right)}$$

可以发现这个过程非常烦琐，如果使用计算机编程会产生 $O(n^3)$ 级别的时间复杂度。而这仅仅是在系统中只有四个结点的情况，随着网络规模的增大，其实这是一个 $O(n^{n-1})$ 级别的问题。

变量消元法用对隐藏变量（本例中为 b 和 c）进行逐个消元的方式简化了这种运算，从编程的角度出发它就是常说的动态规划。对于上述计算问题，用变量消元的步骤可以是：

◎　首先消除变量 c，设函数 $f(a,b)=\sum_c P(c|a,b)P(d=1|b,c)$，原等式变为

$$\frac{\sum_b(P(a)\cdot P(b|a)\cdot f(a,b))}{\sum_b\sum_a(P(a)\cdot P(b|a)\cdot f(a,b))}$$

◎　再消除变量 b，设函数 $g(a)=\sum_b P(b|a)\cdot f(a,b)$，原等式变为：

$$\frac{P(a)\cdot \mathrm{g}(a)}{\sum_a(P(a)\cdot g(a))}$$

这样等式完全变为了变量 a 的函数。将这样的步骤应用在计算机编程中，其时间复杂度就是 $O(n)+O(n)+O(n)$，如果网络规模增大时间复杂度也只是 $n\cdot O(n)$，比原始的链式法则计算公式的指数级别复杂度有显著的减少。

说明：不只是针对计算机编程，即使是手动计算，变量消元法也能显著提高效率。

7.2　网络构建

上一节从意图和理论基础方面学习了贝叶斯网络，本节描述如何在特定的领域中建立这样的贝叶斯网络。构建一般需要分三步进行：确定是领域中有哪些重要的变量及其分布类型（二项分布、多项式分布、高斯分布等）、确定 DAG 图结构、学习网络中每个结点 CPD 中的参数。第一项工作主要由领域专家完成；第二项既可以由专家确定也可以通过数据训练获得；第三项即是有监督学习中的参数估计，主要由数据训练完成参数。

7.2.1 网络参数估计

如果已经确定了网络的 DAG 结构和每个结点的概率分布函数类型，则可通过参数估计学习概率函数中的参数。精确的（exact）网络参数估计的典型代表是最大似然度估计（Maximum Likelihood，ML）和最大后验估计（Maximum a Posterioi，MAP）。此外，下一节中的近似推理也被用于在无法承受精确推理时间要求的情况下来近似估计网络参数。

1. 最大似然估计

在第 3 章中已经学习了贝叶斯公式中似然度的概念，它是指贝叶斯公式 $P(A|D)=\dfrac{P(D|A)P(A)}{P(D)}$ 中的 $P(D|A)$。在贝叶斯网络中，此处的数据 D 就是所有训练数据集，而 A 就是使用的 CPD 参数。因此最大似然估计就是求如下优化目标中 A 的值：

$$\arg\max_A P(D|A)$$

由于有监督学习中假设所有数据样本是被独立采样的，因此该目标等于：

$$\arg\max_A \prod_{d\in D} P(d|A)$$

其中 d 是每一个独立样本。对于似然函数是多项式分布的离散变量，参数组 A 的值可以直接通过样本中各类型样本占样本总数的比例解得。对于符合高斯分布等的连续变量，这样的无条件求极值问题可设原函数偏导方程组为零并求解参数组 A。而在计算机中，为了降低计算时间复杂度，通常使用梯度下降的方法求近似解。

注意：关于梯度下降算法可以回顾第 3 章。

2. 最大后验估计

ML 方法完全凭数据说话，没有办法加入任何人类的已有经验。如果能够获得的训练数据非常有限无法代表数据整体，这就成了 ML 的最大缺点。而最大后验估计 MAP 方法则把优化目标由似然函数变为了后验函数，通过人为输入先验 $P(A)$，MAP 的优化目标是：

$$\arg\max_A P(A|D)=\frac{P(A|D)P(A)}{P(D)}$$

由于无论 A 取值如何都不影响 $P(D)$ 的值，因此该公式等价于：

$$\arg\max_A P(A|D)\cdot P(A)$$

为了便于计算，通常对上式取 log，拆分概率分布的乘法，最终公式变为：

$$\arg\max_{A} \log P(A \mid D) + \log P(A)$$

对目标的求值方法则与 ML 完全一样，可以用数学推导或梯度下降的方法求解。

7.2.2　启发式搜索

如果网络结构不确定，就需要从数据中学习 DAG 网络结构。学习该结构的出发点非常简单，就是搜索所有可能的有向图结构，找出分值最高的结构作为结果，即目标：

$$\arg\max_{G} \mathrm{Score}(G)$$

其中 Score 是对有向图 G 的打分函数。打分函数有很多种形式，比如 Log-Likelihood（LL）、Minimum Description Length（MDL）、Bayesian Dirichlet（BD）等，它们的作用都是评估一个网络与给定数据的拟合程度。对打分函数比较详细的分析可参考 Alexandra M. Carvalho 的论文 *Scoring functions for learning Bayesian networks*，而本节着重介绍如何搜索可能的有向图结构。

1. 穷举遍历有向图是 NP-hard 问题

遍历所有可能的图形当然是可能的，其方法是逐个挑出每个结点与已搜索的结点组合，然后递归完成搜索。学习过图论的读者应该了解，用这种方法遍历所有 DAG 图是一个 NP-hard 问题，即这样问题的时间复杂度过高不适用于解决通用问题。更加让人失望的是，P.Dagum 和 M. Luby 在他们的论文 *Approximating probabilistic inference in Bayesian belief networks is NP-hard* 中指出，求这样问题的近似解也是 NP-hard 问题。

2. 启发式搜索（Heuristic Search）原理

由于穷举所有可能的策略实现性不强，现在对构建贝叶斯网络结构的主流做法是用启发式搜索的方法。所谓启发式搜索也是逐个排查可能图形的方式，与穷举法不同的是它不机械地查找所有可能值，而是采取“走一步看一步”的策略，在每一步用评估函数 $f(G_n)$ 挑选出下一个候选图形中哪一个是最优选择，如图 7-5 所示。

图 7-5 中示意了启发式搜索中连续的两个迭代，分别列在左右两个区域，每个区域的中心点为当前迭代建立的图形。在每个迭代中用评估函数 $f(G_n)$ 测评所有可能的下一种选择，取出最优后继，接下来在下一迭代中以该后继为出发点，再继续评估。如此重复，直

到将所有结点加入 DAG 中。

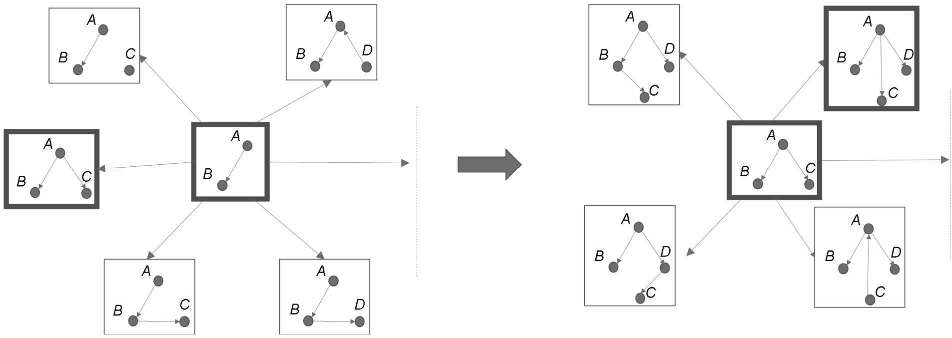


图 7-5　启发式搜索

相对广度优先的穷举搜索法，启发式搜索在每一层只选择一个后继进行下一步搜索，大大缩减了搜索空间。当然，启发式搜索不确保能找到全局最优解，但在配置了合适的评估函数时通常会获得很好的效果。

3. 启发函数与启发搜索策略

按照“走一步看一步”的具体策略不同，常见的启发式搜索又分为贪婪（greedy）与 A*策略两种。它们的根本不同在于采用了不同的评估函数形式：

在贪婪策略中，评估函数 $f(G_n)$ 等于启发函数 $h(G_n)$，启发函数是在启发式搜索中衡量某个状态与目标状态之间距离的函数。启发函数是一个估计值，无须精确计算距离，但一般不应该估计得比真实距离高。如果启发函数高估了该距离，则此启发函数是不可接受的（inadmissible）。

而 A*策略的评估函数由两部分组成，即 $f(G_n)=g(G_n)+h(G_n)$，其中 $h(G_n)$ 仍然是启发函数，而 $g(G_n)$ 被称为成本函数。成本函数是衡量从搜索出发点到当前状态的距离，$g(G_n)$ 是实实在在产生的成本，而不是一个估计值。

从直观角度比较两者的区别：贪婪策略只需对未来的估计选择下一状态，而 A*同时兼顾已经发生的成本和对未来的估计。

7.2.3　Chow-Liu Tree 算法

启发式搜索虽然不再盲目地穷举所有可能图形，但其待搜索的图空间仍然没有变化，

同时启发式搜索也不能保证寻找到最优解。而另一种为了加快组网过程而产生的近似求解策略是缩减可能图的候选空间，即只寻找“树形”的贝叶斯网络结构。

所谓的树形贝叶斯网络是 DAG，它可看成 DAG 的子集，它要求即使将有向图原样变为无向图后，图中仍不存在“环”结构，如图 7-6 所示是对树形网络的举例。

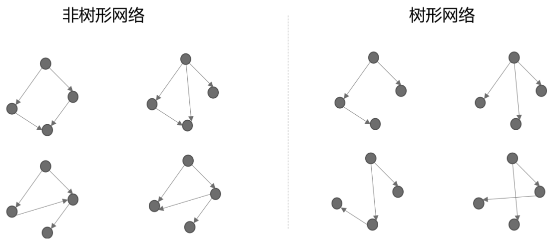


图 7-6　树形网络举例

从图 7-6 中可见，即使是只有四个结点的情况下，待搜索的 DAG 空间也是非常庞大的，而如果仅仅搜索其中的树形结构则会节省大量时间。

Chow-Liu Tree 算法是搜索树形网络空间的典型算法，其主要原理是从空树开始逐个加入结点，而加入的条件是随着该结点加入后产生的边（edge）的权重（weight）最大。边的权重被定义为两个结点的互信息（Mutual Information），即

$$I(x_i;x_j)=\sum_{x_i,x_j}P(x_i,x_j)\log\frac{P(x_i,x_j)}{P(x_i)P(x_j)}$$

其中 x_i 和 x_j 是边两端的结点变量，$I(x_i;x_j)$ 整个公式的作用是衡量两个随机变量的相似程度。在用这种方法确定了所有的 edge 后选取任意结点作为根结点并逐级生成树即可。

在 Chow-Liu 算法的论文中证明了用这种寻找最大权重生成树的方法最终得到的树结构的联合分布与真实样本联合分布最相近。论文中衡量两个分布的方法是 Kullback-Leiber 散度，该指标曾在第 5 章的 t-SNE 中出现过。

7.3　近似推理

第 1 节中已经学习了基于链式法则和变量消元（VE）的推理，但 VE 推理中的大量积分（对于离散变量则是求和）运算使得推理在大规模复杂网络中的计算时间变得不可接受。同样，在确定网络结构后从数据样本训练网络参数的 ML 和 MAP 也有类似问题。因此计

算机科学家只能寻找一些近似推理（Approximate Inference）算法，虽然不能保证找到理论最佳解，但在推理时间和效果上都变得可以接受。

这样的贝叶斯网络近似推理算法可以分为随机方法（Stochastic）和确定性方法（Deterministic）两种，马尔可夫链蒙特卡洛（Markov chain Monte Carlo，MCMC）和变分贝叶斯（Variational Bayes，VB）分别是它们的代表。

7.3.1 蒙特卡洛方法

蒙特卡洛（Monte Carlo）是一种用随机模拟的方式实现计算目的的方法。随机模拟计算方法来自于 18 世纪用于计算圆周率π值的实验，该实验随机向一个方形区域投针，然后用被投入内嵌圆中针的数量来计算圆周率，其计算公式是：

$$\frac{\text{内嵌圆面积}}{\text{方形面积}} \approx \frac{\text{圆内针数}}{\text{方形内针数}} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}$$

因此有 $\pi = 4 \times \frac{\text{圆内针数}}{\text{方形内针数}}$。如图 7-7 所示是该实验在三次不同投针总数情况下的示意图。

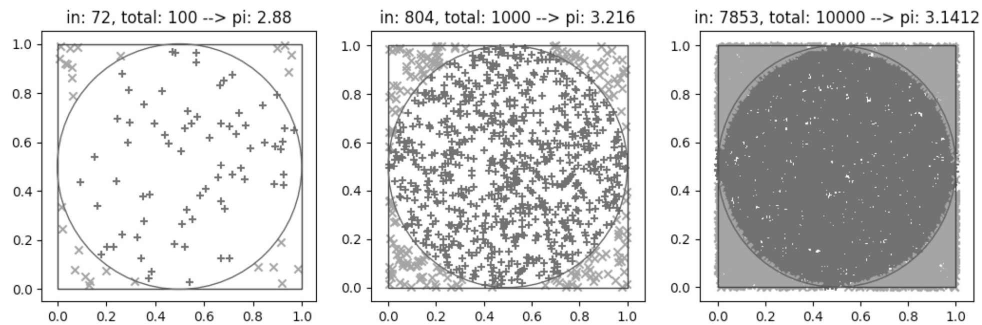


图 7-7　三次投针实验

三次实验的投针总数分别为 100、1000、10000，得到的π值分别为 2.88、3.216、3.1412。可以发现随着投针数（随机事件）的增加，计算结果越来越趋于真实值。

显然，现在即使是小学生也已经无须用这种方式计算圆周率了，但蒙特卡洛方法本身却被启发用于解决各种越来越复杂的问题，比如计算图 7-8 中四个圆形覆盖区域的面积。

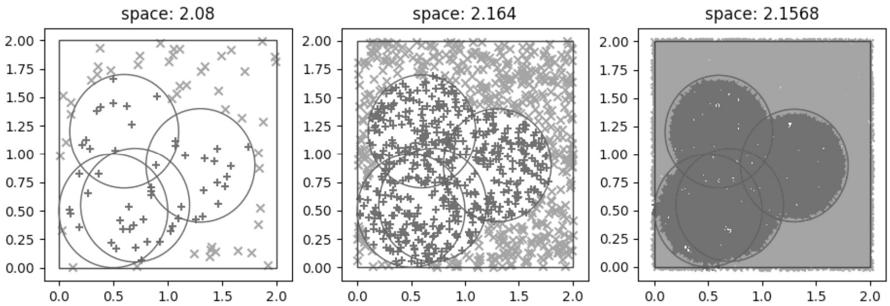


图 7-8　蒙特卡洛方法计算面积

由于外围矩形区域面积已知，只要计算区域内投针所占整体投针的比值就可计算出任意不规则区域的面积。

7.3.2　马尔可夫链收敛定理

在第 6 章中读者已经了解到，马尔可夫链是一种状态链，其中每个结点的状态值概率只依赖于它的紧前结点而与其他结点无关。其中每个状态在下一结点变换为其他状态的概率被称为“转移概率”（Transform Prob），此外为了约束该链中第一个结点的状态还定义了“初始概率”（Initial Prob）的概念。而第 6 章在隐马尔可夫模型的训练中通常不太关注初始概率，而只关心转移概率。这是为什么呢？

马氏链收敛定理可以说明该原因：如果一个非周期马氏链具有转移概率矩阵 $\boldsymbol{P}$ 且它的任何两个状态是连通的，那么 $\lim_{n\to\infty} P_{ij}{}^{n}$ 存在且与 i 无关，记平稳概率分布（stationary distribution）$\lim_{n\to\infty} P_{ij}{}^{n} = \pi\left(j\right)$。

对这段比较学术的定义通俗的解释就是：在一个马尔可夫链上总会找到结点 n，从该结点开始每个后续结点的状态概率分布固定，只与“转移概率”有关而与“初始概率”无关。下面是一段验证马氏链收敛定理的代码：

```
import numpy as np

init_probs = np.array([0.1, 0.3, 0.6])              # 初始概率矩阵
transform_probs = np.array([[0.7, 0.1, 0.2],        # 转移概率矩阵
                            [0.2, 0.5, 0.3],
                            [0.2, 0.4, 0.4]])

chain = [init_probs]                                # 马尔可夫状态概率分布链
```

```
def check_stable():                                    # 检查 chain 变量是否已经平稳
    if len(chain)<3:
        return False
    for i in range(-2, -1):
        if not np.array_equal(np.around(chain[i], 2),
           np.around(chain[i-1], 2)):
            return False
    return True

for i in range(10000):
    chain.append(chain[-1].dot(transform_probs))    # 生成新的结点
    print("iter %s: %s"%(i, np.around(chain[-1], 2)))
    if check_stable():                                 # 如果进入平稳状态则退出
        print("stabled!")
        break
```

以上代码定义了一个有三种状态的马尔可夫链，在给定初始概率和转移概率的情况下逐个生成链上的结点，直到发现最近的三个结点概率分布完全固定为止（此时进入马尔可夫链平稳状态）。上述代码的执行结果如下：

```
iter 0: [ 0.25  0.4   0.35]
iter 1: [ 0.32  0.36  0.31]
iter 2: [ 0.36  0.34  0.3 ]
iter 3: [ 0.38  0.33  0.29]
iter 4: [ 0.39  0.32  0.29]
iter 5: [ 0.4   0.31  0.29]
iter 6: [ 0.4   0.31  0.29]
iter 7: [ 0.4   0.31  0.29]
stabled!
```

可以发现从 iter 5 开始结点的状态概率分布已经不再变化。如果修改代码中的 init_probs 变量为任意其他分布（当然必须满足初始概率之和等于 1）并再次运行代码，会发现该过程最后仍然收敛于相同分布，比如设置 init_probs=[0.7, 0.3, 0.0]时的运行结果如下。

```
iter 0: [ 0.55  0.22  0.23]
iter 1: [ 0.47  0.26  0.27]
iter 2: [ 0.44  0.28  0.28]
iter 3: [ 0.42  0.3   0.28]
iter 4: [ 0.41  0.3   0.29]
iter 5: [ 0.4   0.31  0.29]
```

```
iter 6: [ 0.4   0.31  0.29]
iter 7: [ 0.4   0.31  0.29]
stabled!
```

最终马尔可夫链的结点状态分布仍然收敛于[0.4　0.31　0.29]。但是如果修改 transform_probs 值，马尔可夫链则会收敛于一个不同的分布。这就是马尔可夫链收敛定理的直观效果了：多次迭代后，状态收敛于一个只与转换概率相关的分布，而与初始结点概率分布无关。

7.3.3　MCMC 推理框架

用蒙特卡洛方法进行贝叶斯推理的基本想法是，在贝叶斯网络所表达的联合分布上进行若干次采样，然后用采样后的样本通过参数估计的方法解决推理问题。比如当需要求某离散型变量网络中的概率 $P(x=1, y=0)$时，只需要采样若干个样本（比如 10 个），然后求其中取到 $x=1$ 并且 $y=0$ 的样本所占采样总数的比例，就可得到 $P(x=1, y=0)$的估计值。这样，蒙特卡洛推理的关键就转移到了“如何进行样本采样”上。

MCMC 就是使用马尔可夫链进行采样的蒙特卡洛过程，该方法由 Metropolis 于 1953 年提出。Metropolis 提出如果能在马尔可夫链上构造一个转换概率矩阵，使它在进入平稳状态后收敛于采样的联合分布，则在该马尔可夫链上每个平稳状态结点都是一个取自目标联合分布的样本点。如图 7-9 所示。

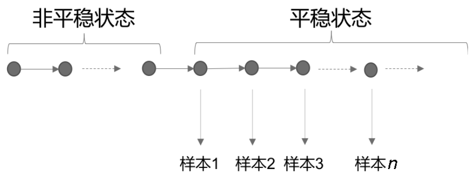


图 7-9　MCMC 采样

在图 7-9 的马尔可夫链中，只要用转移矩阵产生新的结点就能不断地得到新的样本。与上一节中演示马氏链收敛定理的 Python 代码略有不同的是，此时的马尔可夫链上的状态通常是多元状态，因此转移概率分布也会是多元概率分布（比如多元高斯分布、多项式分布等），用这些样本就可以进行参数估计的贝叶斯推理了。

图 7-9 的具体实现有几种不同的算法：Metropolis 的经典算法基于细致平稳条件（Detailed Balance）、接受率（Acceptance Ratio）等；后来 Hastings 改进该算法使得马氏链能够更快收敛，称为 Metropolis-Hastings 算法；而 Stuart 和 Donald Geman 于 1984 年提出

的 Metropolis-Hastings 的一种特殊形式 Gibbs sampling，是目前贝叶斯 MCMC 推理的主流采样算法。

7.3.4　Gibbs 采样

Gibbs 采样的目标是，在不知道确切的联合分布的情况下，仅仅利用条件分布采样到符合目标联合分布的样本。它的本质是构造一个从快速收敛到平稳状态的马尔可夫链。

注意：回忆在贝叶斯网络中每个结点由条件概率分布 CPD 构成，因此在其中获得各种条件分布比获得联合分布容易得多！

例如有某个推理需求是求 $P(x=1, y=1, z=0)$，则需要对联合分布 $P(x, y, z)$ 进行采样。但是此时联合分布 $P(x, y, z)$ 过于复杂难于计算，而条件分布 $P(x \mid y=?, z=?)$ 、 $(y \mid x=?, z=?)$ 、 $P(z \mid y=?, x=?)$ 非常容易获得，则可利用如下步骤进行 Gibbs 采样：

◎　随机定义初始样本 x_0, y_0, z_0（马尔可夫链的初始概率分布并不重要）。

◎　用条件分布 $P(x \mid y=y_0, z=z_0)$ 随机采样一个新的 x 值，记为 x_1；并用 $P(y \mid x=x_1, z=z_0)$ 采样得 y_1，用 $P(z \mid x=x_1, y=y_1)$ 采样得 z_1。因此得到了样本 x_1, y_1, z_1（条件分布起到了马尔可夫链上转移概率的作用）。

◎　重复第二步的采样步骤可以一直产生新的样本（每个样本构成了马尔可夫链上的一个结点）。

上述过程可以构造出一个任意长度的马尔可夫链。而参考图 7-9，只有在平稳状态下的马尔可夫链才可以真正用于 MCMC 采样结果，因此 Gibbs 采样在获取结果时通常跳过最开始的 *n* 个样本（比如 *n*=1000），只将其后的样本作为真正的采样结果，这个过程被称为 burn-in period。

可以将上述三维特征空间的采样过程轻易地扩展到任意高的维度，不同点只在于每个迭代中需要用到更多的条件分布逐个计算每个维度的采样值。

7.3.5　变分贝叶斯

变分贝叶斯（VB）是除 MCMC 外另一种重要的近似推理方法，理解 VB 需要较深的数学功底。泛函是一种描述“函数←→数值”映射关系的工具；变分（Variation）是对泛函的一种处理方法，相当于对函数的微积分处理；变分贝叶斯是用欧拉－拉格朗日方程、

平均场定理等变分工具求解的方法。

在 Matthew J.Beal 于 2003 年发表的博士论文 *Variational Algorithms for Approximate Bayesian Inference* 中对 VB 方法有较全面的描述，有兴趣的读者需要花较多时间研读论文，这里给出一些有用的关于 VB 的概括性描述。

1. 变分贝叶斯的目标

所有的贝叶斯网络推理问题可以归结为某种求条件概率或分布的问题，即 $p(Z|D)$，其中 D 是已观测数据，Z 是待求参数或无法观测的数据。

由于 Z 可以是任意多个事件的组合，所以按照概率公式分解 $P(Z|D)$ 并求值非常复杂：

$$p(Z|D) = p(Z_1, Z_2, \ldots Z_n, | D) = p(Z_1|D)P(Z_2|D, Z_1)\ldots p(Z_n|D, Z_1, Z_n, \ldots)$$

每个分解项都是一个复杂的条件分布。而利用平均场理论（mean field theory），可以找出另一个概率模型 Q 使得：

$$p(Z|D) \approx q(Z) = q(Z_1, Z_2, \ldots Z_n) = q(Z_1)q(Z_2)\ldots q(Z_n)$$

这样就把原本求复杂条件分布的问题转化为了求独立事件的概率问题，因此寻找合适的 Q 分布就成为了 VB 的目标。

2. 用 Kullback-Leibler 寻找 Q 分布

在本书第 5 章的 t-SNE 一节中已经介绍了 Kullback-Leibler 是评估两个概率分布之间相似性的工具。而在 VB 中仍然用该公式衡量 $P(Z|D)$ 与 $Q(Z)$ 的近似程度，根据 Kullback-Leibler 公式的定义，有

$$\mathrm{KL}(qp) = -\int q(Z)\ln\left(\frac{p(Z|D)}{q(Z)}\right)\mathrm{d}Z = \ln(p(D)) - \int q(Z)\ln\left(\frac{p(Z,D)}{q(Z)}\right)\mathrm{d}Z$$

如果设 $\mathcal{L}(q) = \int q(Z)\ln\left(\frac{p(Z,D)}{q(Z)}\right)\mathrm{d}Z$，则有：

$$\ln(p(D)) = \mathcal{L}(q) - \mathrm{KL}(qp)$$

最合适的 Q 分布无疑是使得 $\mathrm{KL}(qp)$ 值最小的 Q 分布，因为 $\ln(p(D))$ 在给定观测数据的情况下固定，因此最小化 $\mathrm{KL}(qp)$ 的任务等同于最大化 $\mathcal{L}(q)$。

对于解决这类优化目标，数学上可以使用在第 5 章中多次提到的拉格朗日乘子法，或

者近似求解算法——EM 算法、梯度上升等。

注意：这里的 $\mathcal{L}(q)$ 定义使用了适用于连续变量的微积分形式，而对于离散变量则退化为求和运算。

由于 $\mathcal{L}(q)$ 中参数 q 是一个概率分布，在数学上 q 本身就是一个函数，则 $\mathcal{L}(q)$ 就是一个泛函，而最大化 $\mathcal{L}(q)$ 的过程就是一个变分过程，这就是变分贝叶斯名称的由来。

3. VB 与 MCMC 的对比

根据 2017 年 12 月 Blei、Kucukelbir 和 McAuliffe 发表的论文 *Variational Inference: A Review for Statisticians*，对 VB 和 MCMC 在推理效果上有如下论断：VB 方法适合于数据量非常大、对推理时间要求高的场景；而 MCMC 适合于数据规模较小、但对推理精度要求较高的场景。也就是说 VB 推理速度更快，而 MCMC 推理效果更好。

7.4 利用共轭建模

在贝叶斯网络中理论上事件的概率分布函数可以是任意形式的，但在实际建模中常用的是几种经典的概率分布，这是为什么呢？

7.4.1 共轭分布

共轭分布（Conjugate Distribution）的概念来源于对简化贝叶斯公式计算的需求。在贝叶斯公式 $P(A|D)=\dfrac{P(D|A)P(A)}{P(D)}\propto P(D|A)P(A)$ 中，如果允许 $P(A|D)$、$P(A)$、$P(D|A)$ 是任何形式的函数，用该公式计算概率分布时会充斥着分解、积分、求和等运算。

对于指数家族（Exponential Family）的概率函数来说则无须这样，它们有一个很好的性质，即在贝叶斯公式计算过程中后验 $P(A|D)$ 和先验 $P(A)$ 具有相同的函数形式，并且在计算过程中也不需要分解所有条件并积分。此时，称 $P(A|D)$ 与 $P(A)$ 的分布函数是似然函数 $P(D|A)$ 的共轭分布；特别地，称先验 $P(A)$ 为共轭先验（conjugate prior）。

用 Binomial 分布（即二项分布）和它的共轭分布 Beta 分布举例说明利用共轭分布进行贝叶斯计算的优势。

◎　Binomial 分布的形式是 $\mathrm{Bin}\left(k;n,p\right)=\begin{pmatrix}n\\k\end{pmatrix}p^{k}(1-p)^{n-k}$，其中 n,p 是超参数。

◎　它的共轭分布是 $\mathrm{Beta}(p;\alpha,\beta)=\dfrac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}p^{\alpha-1}(1-p)^{\beta-1}$，其中 α,β 是超参数；而参数 p 正是对 Binomial 分布中超参数 p 的描述，所以说 Beta 分布可以看成“Binomial 分布之上的分布”。

◎　设 $P(D\mid A)$ 是 Binomial 分布，$P(A)$ 是 Beta 分布。如果没有共轭特性，$P\left(A\mid D\right)\propto P\left(D\mid A\right)P\left(A\right)$ 会是一个具有参数 p 和四个超参数 n、k、α、β 的复杂函数。

◎　由于它们是共轭的，可以知道该乘积也是一个 Beta 分布，实际上在后验分布 $\mathrm{Beta}\left(p;\dot{\alpha},\dot{\beta}\right)$ 中，$\dot{\beta}=\beta+n-k$，这样就带来了计算上的极大便利：

i．随着推理的进行，不会出现超参数无限扩张的趋势；

ii．计算后验概率时无须积分等运算，直接做诸如 $\dot{\alpha}=\alpha+k$、$\dot{\beta}=\beta+n-k$ 这样的简单运算即可。

上述过程省略了计算过程中公式的细节推导，有兴趣的读者可参考概率统计类书籍。在计算过程中似然函数提供的数据被共轭先验“吸收”而生成共轭后验，所以共轭分布有时也被称为“分布的分布”。

除了 Binomial 和 Beta，还有如表 7-1 所示的其他几组常用共轭分布对适用于不同的数据类型。

表 7-1　常用共轭分布对

类型	似然函数分布	共轭分布	说明
离散变量分布	Bernoulli	Beta	单次二值事件实验
	Binomial	Beta	多次二值事件实验
	Poisson	Gamma	单位时间内计数事件实验
	Categorical	Dirichlet	单次多值事件实验
	Multinomial	Dirichlet	多次多值事件实验
连续变量分布	Gaussian	Gaussian	高斯分布
	Multivariate Gaussian	Multivariate Gaussian	多元高斯分布
	Exponential	Gamma	时间间隔事件实验

对表中分布所能描述的事件解释如下。

◎ Bernoulli 是单次二值事件实验的结果，比如抛硬币的结果是正面/反面。

◎ Binomial 用于多次二值事件实验，比如抛了 10 次硬币，4 次正面、6 次反面。

◎ Possion 泊松分布是用于表示单位时间内事件发生的次数，比如地铁里五分钟内到达的乘客数量。

◎ Categorical 与 Multinomial 的关系相当于 Bernoulli 与 Binomial 的关系，用于表示多值事件的单次和多次结果。比如 Categorical 可以用于表示一次掷骰子的点数，而 Multinomial 可以用于表示掷了 10 次骰子的结果是[2, 0, 3, 1, 2, 2]，每个数值分别表示每面骰子出现的次数。

◎ Gaussian 是最常用的连续变量建模分布，两个高斯分布相乘的结果是另一个高斯分布，所以其共轭分布也是 Gaussian。用于表示多元连续变量的 Multivariate Gaussian 同理。

◎ Exponential 用于表征时间间隔，比如地铁站中两个先后到达乘客之间的时间间隔。它与 Possion 从不同的角度描述相同的场景，当然它们都可以用于建模除时间外的其他连续变量，比如距离。

注意：“exponential”与“exponential family”是两个不同的名词，上述所有分布都是指数家族（exponential family）的成员，而指数分布（“exponential”）只是其中之一。

7.4.2 隐含变量与显式变量

将共轭分布应用在贝叶斯网络中，就产生了隐含变量（Latent Variable）与显式变量（Manifest Variable）的概念。任何一个独立的不确定性事件可以在贝叶斯网络中用两个变量表示：隐含变量用于表达该事件的先验和后验概率，而显式变量用相对应的似然函数建模。

举个例子，假设需要对如下事件建模：

```
盒子里有 10 块巧克力，有些是黑巧克力，有些是白巧克力。现在取出 2 块巧克力，如何通过这 2 块巧克力的颜色推测盒子里有多少黑巧克力、多少白巧克力？
```

对该问题的分析是：

◎ 在未取出任何巧克力前，对该盒巧克力的颜色情况有个初始估计。这里未给出这方面的任何假设，较安全的估计可以是“黑、白巧克力有相同的数量”。这就是一个 Beta 类型共轭先验—— $P(A)=\text{Beta}(\alpha=50,\beta=50)$，其中 α 与 β 参数相等表

达了两个结果的可能性相同，它们的数值越大表明先验可信度越高（实际上，该先验的物理含义是：做了 100 次实验，其中两种结果各占 50 次）。

◎ 取出 2 块巧克力后，获得了对盒内巧克力分布的似然估计，即 $P(D|A)=\text{Binomial}(n=2)$。

◎ 盒内巧克力分布则是后验 Beta 分布 $P(A|D)\propto P(A)P(D|A)$。

对于这样的事件，建模如图 7-10 所示。



图 7-10　用隐藏变量和显式变量建模

在图中，隐藏变量只负责概率计算，而显式变量负责从外部获取推理的已知条件。因此在一些贝叶斯网络软件包（比如 BayesPy）中也称隐藏变量为概率结点，称显式变量为输入结点。

隐藏变量与显式变量的拆分实际上取代了 7.1 节中每个结点上的条件概率分布，成为了一种新的表达条件概率的方式。该方式适用性更强，因为有多种适用于离散、连续变量类型的共轭分布对。对于更加复杂的模型，可以通过隐藏变量间的运算生成新的隐藏变量，如图 7-11 所示。

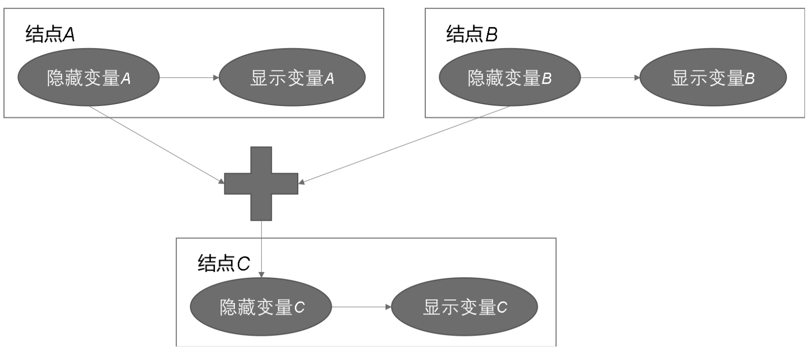


图 7-11　利用共轭分布建模的贝叶斯网络

图 7-11 表达的是一个有三种事件的贝叶斯网络，其中结点 A 和 B 没有父结点，而对它们的隐藏变量进行求和运算后构成了 C 的隐藏变量，也就是它们构成了 C 的条件概率分布。除这里的求和运算外，其他各种初等函数都可成为符合隐藏变量的运算方法。

7.5 实战：胸科疾病诊断

至此已经完成了贝叶斯网络动机和基本原理方面的学习，可以尝试动手开发了。本节用胸科疾病诊断（PyMC3）演示贝叶斯网络推理与采样。

7.5.1 诊断需求

这是一个经典的贝叶斯网络应用场景案例，被用于很多大学的 BN 教案，但这些教案大多只涉及介绍动机与场景而没有给出具体实现。

该案例假设已知贝叶斯网络结构和每个结点的条件概率表，如图 7-12 所示。

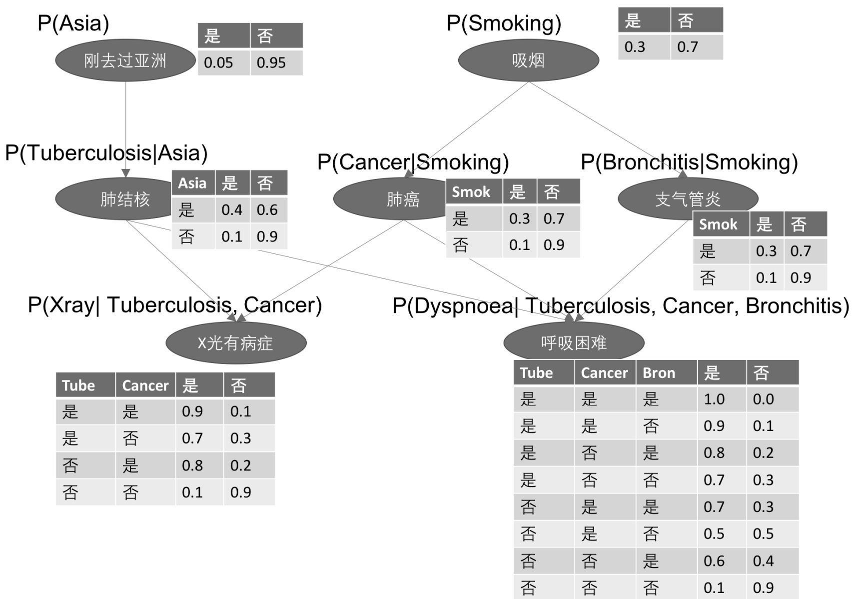


图 7-12 胸科诊断的贝叶斯网络和每个结点的条件概率表

该网络从上到下分为三层。

◎ 第一层是两种生病的原因：最近去过亚洲（假设当时亚洲正流行肺结核疾病）、有吸烟习惯。

◎ 第二层是三种可能的疾病：肺结核、肺癌、支气管炎。

◎　第三层是病人的症状：X-ray 检查是否有异样、呼吸困难。

通过网络中的边可知，是否去过亚洲会影响患肺结核的概率，吸烟会导致肺癌和支气管炎，只有肺结核和肺癌能通过 X-ray 检查到病灶，但三种疾病都会导致呼吸困难。

所有结点都只有是、否两种可能结果，所以该网络中的所有结点均用 Bernoulli 建模。利用该网络希望获得的推理能力是给定任何病因或症状的组合，能够计算出患任意一种疾病的概率，比如：

◎　某个病人去过亚洲，且呼吸困难，则他/她患肺结核、肺癌的概率是多少？

◎　某个病人吸烟、X-Ray 检查结果异常、呼吸困难，他/她患肺癌的可能性是多少？

接下来从工具的选择开始，逐步完成网络建模和推理实践。

7.5.2　Python 概率工具包

由于贝叶斯网络是一个概率统计与机器学习的交叉领域，在 Python 中尚未出现对本章中所有推理与构建算法均能完整实现的工具包，本节介绍其中 4 个目前仍在活跃的项目。

1. BayesPy

BayesPy 是发布于 2013 年并且仍在不断更新的贝叶斯推理工具，它的目标在于为较专业的使用者提供灵活、高效的变分贝叶斯工具。相对于其他工具来说，BayesPy 的调用方式略显复杂。

2. PyMC3

从名称来看，它的开发起始于对马尔可夫链蒙特卡洛（MCMC）的封装。但经过 10 余年的发展它已经是一个功能非常全面的概率统计库，除了基本贝叶斯网络采样和预测，还提供了比较丰富的样例实现贝叶斯模型比较、贝叶斯回归、高斯过程、混合模型等。

3. Edward

这是一个专注于贝叶斯概率统计与推理的工具，它的功能略逊于 PyMC3，但 Edward 是基于 TensorFlow 的工具包，相比 PyMC3 使用的 Theano 可能会获得越来越多的关注。

4. Pomegranate

Pomegranate 是一个封装程度较高的概率库，易用性高于前三者，可以提供类似 scikit-learn 的使用方式。虽然其支持启发式搜索 A*等自动网络构建算法，但由于仅支持对离散类型变量建模，使得该库虽适合初学者入门但实用性不强。

表 7-2 总结了这些工具及它们的功能点。

表 7-2 Python 贝叶斯网络工具包及其功能点

	Bayespy	PyMC3	Edward	Pomegranate
离散变量建模	√	√	√	√
连续变量建模	√	√	√	
变量消元（VE）等准确推理（Exact Inference）		√	√	√
MCMC 推理		√	√	
变分贝叶斯（VB）推理	√	√	√	
自动网络构建				√
支持 Python2		√	√	√
支持 Python3	√	√	√	√

目前还没有哪一个工具是完备的，如何选择取决于功能需求。相对来说，Bayespy 专注于变分贝叶斯推理，PyMC3 与 Edward 是比较全面的推理与采样工具，Pomegranate 同时允许自动网络构建和推理但不支持连续变量是较大的缺陷。

由于在需求中已经给出网络结构和每个结点的条件概率，因此本案例选用目前为止功能最强大的 PyMC3 作为工具实现图 7-12 的贝叶斯网络。在实验前，首先需要确保已经安装了 PyMC3 工具包，其安装命令为：

```
# ../venv/bin/pip3 install pymc3 patsy pandas
```

除 PyMC3 外，patsy 和 pandas 是 PyMC3 的一些高级功能用到的统计工具。如果 PyMC3 依赖的 Numpy、Theano 等尚未安装，该条命令会自动安装它们。

7.5.3 建立模型

本节通过 PyMC3 代码实现图 7-12 的网络建模。由于都是 Bernoulli 事件，图中任何事

件的概率分布用一个数字即可表达，比如对于 P(Asian)来说用 Bernoulli(p=0.05)表示即可，代码如下。

```
import pymc3 as pm                                        #引入 PyMC3 模块

basic_model = pm.Model()
with basic_model:
    # 第一层，病因结点
    asia_p = pm.Beta('asia_p', 5, 95)
    asia = pm.Bernoulli('asia', p=asia_p)
    smoking = pm.Bernoulli('smoking', p=0.3)

    # 第二层，疾病结点
    tuberculosis_p = pm.Deterministic('tuberculosis_p',
                            pm.math.switch(asia, 0.4, 0.1))
    tuberculosis = pm.Bernoulli('tuberculosis', p=tuberculosis_p)
    cancer_p = pm.Deterministic('cancer_p',
                            pm.math.switch(smoking, 0.3, 0.1))
    cancer = pm.Bernoulli('cancer', p=cancer_p)
    bronchitis_p = pm.Deterministic('bronchitis_p',
                            pm.math.switch(smoking, 0.3, 0.1))
    bronchitis = pm.Bernoulli('bronchitis', p=cancer_p)

    # 第三层， 症状结点
    xray_p = pm.Deterministic('xray_p', pm.math.switch(tuberculosis,
                                        pm.math.switch(cancer, 0.9, 0.7),
                                        pm.math.switch(cancer, 0.8, 0.1))
    )
    xray = pm.Bernoulli('xray', p=xray_p)
    dyspnea_p = pm.Deterministic('dyspnea_p',
                        pm.math.switch(tuberculosis,
                            pm.math.switch(cancer,
                                pm.math.switch(bronchitis, 1.0, 0.9),
                                pm.math.switch(bronchitis, 0.8, 0.7)),
                                pm.math.switch(cancer,
                                    pm.math.switch(bronchitis, 0.7, 0.5),
                                    pm.math.switch(bronchitis, 0.6, 0.1))))
    dyspnea = pm.Bernoulli('dyspnea', p=dyspnea_p)
```

分析该代码如下。

◎ 在 PyMC3 中贝叶斯网络模型用 Model 对象表示，用 with 语句引入该对象上下文后即可定义网络内结构。

◎ 每个结点都是二值类型，因此用 pm.Bernoulli 对象定义显示（manifest）变量。每个结点的第一个参数用于告知 PyMC3 该结点的名称，用于在后续的分析中表示该变量。

◎ 定义了 pm.Beta 类型的隐含（latent）变量 asia_p，用于表达变量 asia 的共轭先验。

◎ 显式变量 smoking 没用使用隐含变量，而是使用了固定的概率值 0.3（即吸烟的比率是 30%，不吸烟的比率为 70%）。

◎ 其他结点的隐含变量用 pm.Deterministic 对象定义，该对象用于表示从其他变量值通过确定性计算获得先验概率。比如 pm.Deterministic('tuberculosis_p', pm.math.switch(asia, 0.4, 0.1))表示，当 aisa 是 True 时 tuberculosis 的概率是 0.4，而当 asia 是 False 时 tuberculosis 的概率是 0.1。这些 Deterministic 变量实际描述了不同事件之间的条件概率表。

7.5.4　MCMC 采样分析

PyMC3 的推理主要通过采样、分析两步完成，其中采样默认使用 MCMC 方式，并提供若干文本、可视化工具进行分析。

1. 采样

使用 sample()函数可以直接对模型进行采样，比如：

```
with basic_model:
    trace = pm.sample(100)
```

采样会自动根据网络中的结点分布类型选择合适的采样算法，如图 7-13 所示，本次采样使用 BinaryGibbsMetropolis 作为显式变量的采样算法。

```
Multiprocess sampling (4 chains in 4 jobs)
CompoundStep
>NUTS: [asia_p_logodds__]
>BinaryGibbsMetropolis: [asia, smoking, tuberculosis, cancer, bronchitis, xray, dyspnea]
 99%|██████████████████████████████████████████████████| 595/600 [00:01<00:00, 361.23it/s]
```

图 7-13　PyMC3 采样过程输出

此处对已经定义过的模型 basic_model 进行 100 条采样，并保存在 trace 变量中。变量 trace 是一个 MultiTrace 类型的对象，可以通过它的 points()方法获得轮询采样结果的迭代器，比如：

```
with basic_model:
    for i, s in enumerate(trace.points()):
        print(i, s)
```

返回结果如下：

```
0 {'asia_p_logodds__': -2.3974357200987688, 'asia': 1, 'cancer': 0, 'bronchitis': 0, 'asia_p': 0.083368444775973133, 'tuberculosis_p': 0.40000000000000002, 'cancer_p': 0.29999999999999999, 'bronchitis_p': 0.29999999999999999, 'xray_p': 0.69999999999999996, 'dyspnea_p': 0.69999999999999996}

1 {'asia_p_logodds__': -2.3765916122643262, 'asia': 0, 'cancer': 0, 'bronchitis': 1, 'asia_p': 0.084975208339643366, 'tuberculosis_p': 0.10000000000000001, 'cancer_p': 0.29999999999999999, 'bronchitis_p': 0.29999999999999999, 'xray_p': 0.69999999999999996, 'dyspnea_p': 0.80000000000000004}

2 {'asia_p_logodds__': -2.489089076504456, 'asia': 0, 'cancer': 0, 'bronchitis': 0, 'asia_p': 0.076626624878668961, 'tuberculosis_p': 0.10000000000000001, 'cancer_p': 0.29999999999999999, 'bronchitis_p': 0.29999999999999999, 'xray_p': 0.69999999999999996, 'dyspnea_p': 0.69999999999999996}
…
```

这里列出了前 3 条采样值，每一条都是一个 dictionary 对象明确地表示每个结点的采样值。

2. 分析采样

模型开发者关心的往往不是单条的采样数据，而是采样总体的统计结果。可以用 summary()函数获得该统计：

```
with basic_model:
    print(pm.summary(trace))
```

该段代码输出如图 7-14 所示的结果。

	mean	sd	mc_error	hpd_2.5	hpd_97.5	n_eff
asia	0.057500	0.232796	0.010521	0.000000	1.000000	123.0
smoking	0.295000	0.456043	0.018835	0.000000	1.000000	400.0
tuberculosis	0.132500	0.339034	0.021925	0.000000	1.000000	154.0
cancer	0.142500	0.349562	0.022151	0.000000	1.000000	190.0
bronchitis	0.160000	0.366606	0.019209	0.000000	1.000000	347.0
xray	0.282500	0.450215	0.030752	0.000000	1.000000	147.0
dyspnea	0.275000	0.446514	0.026339	0.000000	1.000000	171.0
asia_p	0.051785	0.024954	0.001802	0.010799	0.098227	151.0
tuberculosis_p	0.117250	0.069839	0.003156	0.100000	0.400000	123.0
cancer_p	0.159000	0.091209	0.003767	0.100000	0.300000	400.0
bronchitis_p	0.159000	0.091209	0.003767	0.100000	0.300000	400.0
xray_p	0.269250	0.290997	0.020913	0.100000	0.800000	126.0
dyspnea_p	0.298750	0.265091	0.017746	0.100000	0.700000	143.0

图 7-14　采样结果输出

对每个变量都列出了如下值。

◎ mean：均值，即预测值。如果没有任何观测值，该值与网络中的先验概率相近。

◎ sd：standard deviation，衡量变量数据值的集中程度。

◎ mc_error：Monte Carlo error，蒙特卡洛统计的可信度，样本数越多该值越小（越好）。

◎ hdp：highest density region，样本最集中值域的长度。hpd_2.5 是最集中的 2.5%样本，hpd_97.5 是最集中的 97.5%样本，该值一般与 sd 有反向关联。

◎ n_eff：the effective sample size，有效样本数。

3. 采样结果可视化

PyMC3 还提供了可视化分析采样结果的工具，比较典型的是 traceplot()，其使用方法如下。

```
import matplotlib.pyplot as plt

with basic_model:
pm.traceplot(trace, varnames=[ 'xray_p', 'xray'])
plt.show()
```

其中 varnames 是可选参数，用于指定需要绘制的变量。如不指定该参数，traceplot()绘制网络中的所有变量。本例中只绘制 xray_p 和 xray 两个变量，如图 7-15 所示。

在该图中，左侧是对样本总数的统计，本例中 xray 在 0.0 附近的值较高，即说明大多数样本的 xray 结果为阴性/正常。右侧是对每个样本真实值的绘制，出现交错的四条线是

因为 PyMC3 默认使用四个马尔可夫链进行采样，该值可在 sample()函数中进行配置。

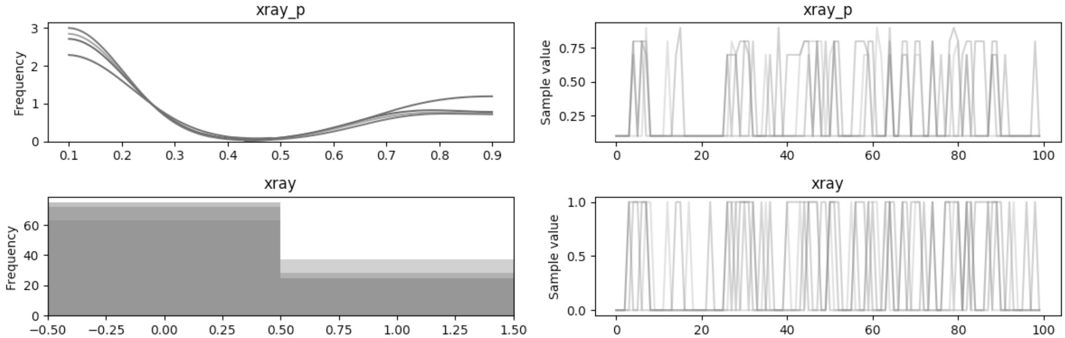


图 7-15　采样结果可视化

7.5.5　近似推理

在 PyMC3 中，推理是输入已观测到的数据（observed data）并对模型进行采样和分析的过程。已观测数据通过结点显式变量的 observed 属性进行配置，比如：

```
with basic_model:
    # 省略其他变量定义
    asia = pm.Bernoulli('asia', p=asia_p, observed=False)
    smoking = pm.Bernoulli('smoking', p=0.3, observed=True)
    xray = pm.Bernoulli('xray', p=xray_p, observed=True)
    dyspnea = pm.Bernoulli('dyspnea', p=dyspnea_p, observed=True)
```

以上代码定义的是一组已知变量：未去过亚洲、有吸烟习惯、X 光呈阳性且有呼吸困难。通过如下采样与分析代码可完成推理：

```
with basic_model:
    trace = pm.sample(1000)
    print(pm.summary(trace, varnames=['tuberculosis', 'cancer',
                         'bronchitis']))
```

输出如图 7-16 所示。

	mean	sd	mc_error	hpd_2.5	hpd_97.5	n_eff	Rhat
tuberculosis	0.3225	0.467433	0.026994	0.0	1.0	273.0	1.007340
cancer	0.7225	0.447765	0.022331	0.0	1.0	356.0	1.004233
bronchitis	0.4000	0.489898	0.014577	0.0	1.0	400.0	0.996094

图 7-16　疾病推理输出结果

可见该样本患 cancer 的概率已经非常高，并且伴有相当程度的 tuberculosis 与 bronchitis 可能。

除了用 MCMC 采样，PyMC3 还支持变分贝叶斯采样，调用方法为：

```
with basic_model:
    approx = pm.fit()
trace = approx.sample(1000)
```

对 trace 变量的分析方法与之前 MCMC 采样的分析方法一致，这里不再重述。变分贝叶斯计算速度较 MCMC 更快，适合网络较复杂的连续类型分布推理。

7.6 本章内容回顾

◎ 贝叶斯网络（BN）是一种概率推理的终极工具，它是一个无环有向图。

◎ BN 中每个结点是对条件概率分布的表述，而整个网络是对联合概率分布的表述。

◎ 联合概率、边缘概率、条件概率的换算关系。

◎ 用变量消元法（VE）可以进行确切（exact）推理。

◎ 在已知网络 DAG 结构的情况下，可以用最大似然估计、最大后验估计等进行确切的网络参数估计。

◎ 在网络结构未知的情况下自动检测网络结构是一个 NP-hard 难题，目前只能用启发式搜索、树模型等进行近似拟合。

◎ 近似推理主要有 MCMC 和 VB 两种算法，这两种方法也用于网络参数估计。

◎ 蒙特卡洛方法是一种用随机模拟的方式实现近似计算目的的方法。

◎ 任何非周期马尔可夫链最终收敛于稳定的状态概率分布。

◎ Gibbs 采样是一种典型的 MCMC 采样法，它是 Metropolis-Hastings 的一种特例。

◎ 变分贝叶斯用 Kullback-Leibler 寻找与目标分布近似的 Q 分布，该分布可以加快推理速度。

◎ MCMC 与 VB 的比较：MCMC 更准确，VB 更快。

◎ 共轭分布简化了贝叶斯网络中的概率计算，常见的共轭分布对是 Bernoulli 与 Beta、Binomial 与 Beta、Categorical 与 Dirichlet、Mulitomial 与 Dirichlet、Poisson 与 Gamma、Gaussian 与 Gaussian。

◎ 共轭分布常用于为 BN 中的隐含变量建模。

◎ PyMC3 是 Python 中功能较全面的概率统计与机器学习工具包。

◎ 在 PyMC3 中用 sample()函数获得 MultiTrace 类型的采样结果，之后可以通过多种文本或图形化的方式分析采样结果并完成推理。

第 8 章 自然语言处理

自然语言处理（Natural Language Process，NLP）是当前机器学习应用较多的一个领域，典型的案例有聊天机器人、文本聚类与情感分析等，此外源于 NLP 的 Word2vec 等模型还被应用于在线推荐等非 NLP 领域。与之前章节不同，本章不是对某一类机器学习模型的探讨，而更倾向于介绍当前活跃的工业界 NLP 应用是如何基于机器学习技术建模的，因此本章会有更多的实践，主要内容如下。

◎ 文本建模：基于词袋模型的文章关键字提取、相似度分析等。

◎ 词处理：中文分词、用 Word2vec 寻找近义词等。

◎ 主题模型：比较 NMF、LSA、PLSA、LDA 等技术，建立“文档-主题-单词”的三层模型。

◎ 案例：TF-IDF 关键词推举、Word2vec 寻找近似词、主题模型应用与调参。

8.1　文本建模

本节关注以整段文本或整篇文章作为处理对象的 NLP 方法，包括词袋建模、用 TF-IDF 提取关键字等。这些方法不仅能应用于自然语言，同样可应用于日志分类等其他类型的文本处理。

8.1.1　聊天机器人原理

以文本作为有监督或无监督学习的特征来源，可以完成很多有意义的应用。比如用有监督学习模型可以完成购物网站评价短语的情感判别，用无监督的聚类可以在搜索引擎中归类相似文章、避免重复推荐等。

当前一些在线机器人可以完成查天气、推荐歌曲等任务，而购物网站上的客服机器人可以自动回答一些关于产品物流、售后等相关问题。其实这些常见的在线聊天机器人也多是通过对文本进行有监督学习完成的，如图 8-1 所示。

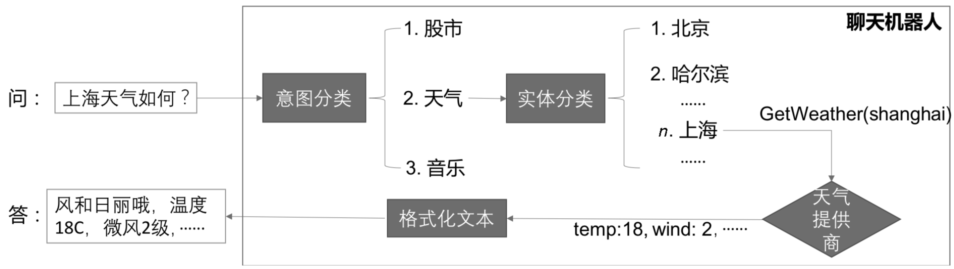


图 8-1　聊天机器人原理

在图 8-1 中的机器人能够正确做出反馈的关键因素是两个分类器：意图（intent）分类器、实体（entity）分类器。机器人收到一条语句后，首先用意图分类器确定使用者发送该条消息的目的是什么，而后再用实体分类器提取意图中的变量，接下来就可以用结构化方法做后续的逻辑处理和反馈了。

这些分类器是如何得到的呢？以意图分类器举例，用本书第 3 章中的任何模型都可以训练出该分类器。

◎　首先需要预先定义机器人所能处理的问题类型（即有监督学习标签空间），如股票查询、天气查询、音乐推荐等。

◎ 收集每一个意图对应的训练数据集，比如“今天下雨吗？”“天气怎么样？”“天气预报”这些文本是把“天气”这个意图作为标签的训练数据。

◎ 将所有“文本→意图”数据集作为训练数据，适配出分类器。

实体分类器会略微复杂，需要通过分词或深度学习模型进行有监督学习。除了这些，工业界的聊天机器人没有任何更加“智能”的成分了，它们所依赖的就是特定领域内丰富的意图与实体的文本集。在 NLP 项目的最开始，首先要解决的是如何将自然语言文本输入各种机器学习模型中的问题。

8.1.2 词袋模型

机器学习中所有模型的输入与输出都是有限长度的数值向量，因此对于自然语言这类没有固定长度的文本来说，处理它们的第一个任务就是如何将文本数字向量化。

词袋模型（Bag-of-Words，BoW）就是一个从文本提取特征转化为数值向量的手段。虽然文本可以是任意长度的，但是任何语言的词汇量都是有限的。词袋模型的输入是一整段文本，而输出是一个词汇计数向量，如图 8-2 所示。

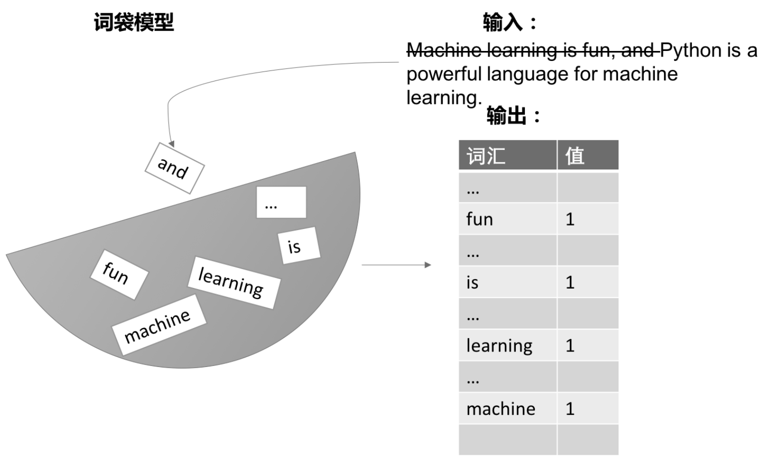


图 8-2　词袋模型

可以想象该模型是一个袋子，将文本中的所有单词包装在了一起。因此句子中原有的单词顺序丢失了，输出的只是每个单词的重量（计数）。

显然词袋模型是对无法对单词排列进行精确建模的一种妥协，但它至今被广泛应用，

其动机来自于一个基本假设：文本的含义取决于构成该文本的单词，而单词的顺序往往不太重要。比如，一篇谈论美食的文章很有可能由“鱼”“鲜美”“炒”“煎”这类单词构成，而“机器学习”“贝叶斯”“编程”等词很少会出现在它们之中。

2. 应用

利用词袋模型将文本转换为数值特征后，就可以使用之前介绍的任何模型处理文本了。比如要对如下三段文本进行聚类：

```
a). Machine learning is fun.
b). Machine learning can analysis business.
c). I like fishing.
```

首先通过词袋模型对它们建模后可以转换成表 8-1 所示的特征向量。

表 8-1　文本转换成特征向量

	machine	learning	is	fun	can	analysis	business	I	like	fishing
a)	1	1	1	1	0	0	0	0	0	0
b)	1	1	0	0	1	1	1	0	0	0
c)	0	0	0	0	0	0	0	1	1	1

此时再用第 4 章中的任何聚类算法对表中的三个向量进行聚类，一定可以把 a)和 b)两个文本成功地聚类在一起。而如果给每条文本都赋予一个意图（intent）标签，则对特征向量分类即可实现上一节中介绍的聊天机器人意图分类器。

3. 停用词（Stop Word）

在自然语言处理领域，停用词是一个常见的概念，它用于过滤一些本身无法表达任何意义的单词，通常是一些介词、代词。比如在表 8-1 的词袋中，可以把 is、I 这样的词设为停用词，这样可以减少词袋的规模，提高后续处理的效率和准确性。

8.1.3　访问新闻资源库

在继续本章的学习之前，建议读者先学习使用 scikit-learn 的“20newsgroups”文本资源库，该库中有上万条普通的自然语言邮件，它们会是本章后续实践的文本资料库。

1. 下载资源库

首先可以通过如下命令下载该文本库：

```
>>> from sklearn.datasets import fetch_20newsgroups
>>> twenty_train = fetch_20newsgroups(subset='train')
```

第一次访问该资源库时需要耐心等待上述调用返回，它会从互联网下载文本库。该过程中如果出现如下 SSL 的错误：

```
ssl.SSLError: [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed
(_ssl.c:645)
```

可以寻找计算机中的“Install Certificates.command”文件，运行后可以解决该问题。

2. 资源预览

“20newsgroups”是一个已经分类了的资源库，这意味着每条文本都已经有了一个标签。可以通过列表属性 data 和 target 属性访问文本和对应标签：

```
>>> len(twenty_train.data)                                # 查看文本的数量
11314

>>> len(twenty_train.target)                              # 查看标签的数量
11314

>>> twenty_train.data[0]                                  # 显示第一条文本内容
"From: lerxst@wam.umd.edu (where's my thing)\nSubject: WHAT car is
this!?\nNntp-Posting-Host: rac3.wam.umd.edu\nOrganization: University of
Maryland, College Park\nLines: 15\n\n I was wondering if anyone out there could
enlighten me on this car I saw\nthe other day. It was a 2-door sports car, looked
to be from the late 60s/\nearly 70s. It was called a Bricklin. The doors were
really small. In addition,\nthe front bumper was separate from the rest of the
body. This is \nall I know. If anyone can tellme a model name, engine specs,
years\nof production, where this car is made, history, or whatever info you\nhave
on this funky looking car, please e-mail.\n\nThanks,\n- IL\n   ---- brought to
you by your neighborhood Lerxst ----\n\n\n\n\n"

>>> twenty_train.target[:10]                              # 显示前 10 个标签
array([ 7,  4,  4,  1, 14, 16, 13,  3,  2,  4])
```

因为列表 data 与 target 中的元素一一对应，所以它们的长度相同。当前该新闻库中有

11314 条文本资源。

3. 标签含义

在 target 属性中所有的标签都用数字表示，其实它们是有英文名称的。可以用属性 target_names 获得它们的英文名称：

```
>>> twenty_train.target_names
['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc',
'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware', 'comp.windows.x',
'misc.forsale', 'rec.autos', 'rec.motorcycles', 'rec.sport.baseball',
'rec.sport.hockey', 'sci.crypt', 'sci.electronics', 'sci.med', 'sci.space',
'soc.religion.christian', 'talk.politics.guns', 'talk.politics.mideast',
'talk.politics.misc', 'talk.religion.misc']
```

通过该列表可以知道：target==0 的标签含义为 alt.atheism，target==1 的标签含义为 comp.graphics，以此类推。

4. 格式化显示文本

通过 data 属性直接显示的文本资源非常杂乱，可以用简单的 Python 技巧对它们进行格式化，代码如下。

```
>>> print("\n".join(twenty_train.data[0].split("\n")))
From: lerxst@wam.umd.edu (where's my thing)
Subject: WHAT car is this!?
Nntp-Posting-Host: rac3.wam.umd.edu
Organization: University of Maryland, College Park
Lines: 15

 I was wondering if anyone out there could enlighten me on this car I saw
the other day. It was a 2-door sports car, looked to be from the late 60s/
early 70s. It was called a Bricklin. The doors were really small. In addition,
the front bumper was separate from the rest of the body. This is
all I know. If anyone can tellme a model name, engine specs, years
of production, where this car is made, history, or whatever info you
have on this funky looking car, please e-mail.

Thanks,
- IL
   ---- brought to you by your neighborhood Lerxst ----
```

如此，这封邮件的内容就非常清晰了，这些文本将是本章后续进行 NLP 处理的具体内容。

8.1.4 TF-IDF

处理大段文本的一个常见任务是从其中提取关键词汇。谷歌、百度这样的搜索引擎可以利用这些关键字提高文章搜索效率，信息监控系统也可以用这类技术从运行日志中查找异常信息。TF-IDF（Term Frequency–Inverse Document Frequency）是一种完成这项任务的工具，而且它是无监督的，即完全不需要人工干预。

TF-IDF 也是一种词袋模型，它的作用是在一个由多个文章组成的文集（corpus）中计算出每个单词对其所在文本的重要程度。因此，TF-IDF 工具的输入必须是整个文集，而输出是一个类似表 8-1 的“文章-单词”矩阵。但矩阵中的数值单元内容与经典的词袋模型不同，每个单元除了蕴涵对单词的计数功能，还需要加入“重要性”信息。

如其命名，TF-IDF 输出矩阵的每个单元由两部分组成，其结果是这两部分的乘积。其中 TF 部分代表词频，公式为：

$$\mathrm{TF}_{i,j}=\frac{n_{i,j}}{\sum_k n_{k,j}}=\frac{\text{单词}i\text{在第}j\text{个文章中出现的次数}}{\text{文章}j\text{中所有单词出现的次数}}$$

也就是说，某个单词越是集中地出现在某篇文章中，该单词对该文章来说 TF 值越高。

TF-IDF 的第二个部分是逆向文件频率（IDF），其公式为：

$$\mathrm{IDF}_{i,j}=\lg\frac{\text{文集中文章总数}}{\text{包含单词}i\text{的文章数量}+1}$$

也就是说，某个单词越广泛地分布于各个文章中，其 IDF 值越低。IDF 的理念有点像多选投票，比如，假设奥斯卡最佳影片的提名榜单中有五个候选影片，并且允许每个评委进行多选投票，如果一个评委给这五个影片都投了票，则该评委的投票对所有影片都没有意义，也就是说一个评委投票越少则其投票越有意义。

这样，将 TF 值与 IDF 值相乘就获得了能表示每个单词在文章中重要性的 TF-IDF 矩阵。

8.1.5 实战：关键词推举

本节通过 scikit-learn 提供的文本特征提取类应用 TF-IDF 技术处理“20newsgroups”

文本库，自动生成所有邮件的关键词。

1. CountVectorizer 类

在 scikit-learn 的 sklearn.feature_extraction.text 包中提供了文本特征提取的相关封装类，其中 CountVectorizer 可以生成词袋模型矩阵，TfidfTransformer 用于生成 TF-IDF 矩阵。与 scikit-learn 中其他功能的调用风格类似，在初始化类对象后调用 fit_transform()函数可获得结果矩阵，代码如下。

```
>>>from sklearn.feature_extraction.text import CountVectorizer

>>>count_vect = CountVectorizer(stop_words='english')      # 生成对象
# 生成词袋模型矩阵
>>>X_train_counts = count_vect.fit_transform(twenty_train.data)
>>> print(type(X_train_counts), X_train_counts.shape)
<class 'scipy.sparse.csr.csr_matrix'> (11314, 129796)
```

转换的输出是一个 11314×129796 的矩阵，说明在 11314 个文章中共有 129796 个词汇，矩阵 X_train_counts 中的值是每个词汇在每篇文章中的出现次数。此外，还可以通过 vocabulary_属性获得词汇表，它是一个“词汇文本→词汇编号”字典，比如：

```
>>>count_vect.vocabulary_.get(‘computer’)
41555
```

这说明 X_train_counts 矩阵的第 41555 列是单词 computer 在各文章中的出现次数。

上述代码在初始化 CountVectorizer 对象时设置了 stop_words= ‘english’，即使用内置的默认英文停止词汇。如果需要自定义停止词，也可以对该参数传入一个字符串列表。此外，CountVectorizer 还支持如下常用初始化参数。

◎ encoding：文本编码方式，默认为 utf-8。

◎ ngram_range：是一个（min_n，max_n）对，用于设置词汇的长度范围。比如，如果设置了 ngram_range=(2, 3)，则对于文本“I like machine learning”将生成如下五个词汇。

i.“I like”

ii.“like machine”

iii.“machine learning”

iv.“I like machine”

v.“like machine learning”

◎ max_df：一个 0～1 的数值，指定词汇出现的文章数上限。比如 0.8 定义“在 80% 以上文章中出现过的单词不被放入词袋模型矩阵中”。本参数用于自动过滤一些普遍性的词汇。

◎ min_df：一个 0～1 的数值，指定词汇出现的文章数下限，作用与 max_df 相反。

◎ binary：在词袋矩阵中不对单词计数，而仅仅用 0/1 表示单词是否出现过。

2. TfidfTransformer 类

TfidfTransformer 类对象用于将普通词袋矩阵转化成归一化的 TF-IDF 矩阵，代码如下。

```
>>>from sklearn.feature_extraction.text import TfidfTransformer # 引入类

>>>X_train_tfidf = TfidfTransformer(use_idf=True).fit_transform (
                                                     X_train_counts)
>>> print(type(X_train_tfidf), X_train_tfidf.shape)       # 生成 TF-IDF
<class 'scipy.sparse.csr.csr_matrix'> (11314, 129796)
```

生成的结果是一个与 X_train_counts 形状相同的矩阵。

3. 搜索关键字

利用上述调用并设置适当参数，可以自动提取“20newsgroups”中每一封邮件的关键字，完整代码如下。

```
import numpy as np
from sklearn.datasets import fetch_20newsgroups
twenty_train = fetch_20newsgroups(subset='train')         # 读取邮件数据库

from sklearn.feature_extraction.text import CountVectorizer
count_vect = CountVectorizer(stop_words='english',
                       max_df=0.8, min_df=0.001)

# 仅处理邮件内容，过滤 From、Posting-Host、Organization 等邮件元信息
email_data = ["\n".join(email.split("\n")[5:]) for email in
             twenty_train.data]
X_train_counts = count_vect.fit_transform(email_data)      # 词袋模型矩阵
```

```
dict_word = {count_vect.vocabulary_[key]: key for key in
             count_vect.vocabulary_.keys()}
# 词汇数组
words = np.array([dict_word[idx] for idx in range(len(dict_word))])

from sklearn.feature_extraction.text import TfidfTransformer
# TF-IDF
X_train_tfidf = TfidfTransformer(use_idf=True).fit_transform (
                                                  X_train_counts)

def get_keywords(doc_index):                      # 提取单个邮件的关键字
   KEYWORD_COUNT = 5
   tfidf_row = X_train_tfidf.getrow(doc_index).toarray().flatten()
   # 查找 TF-IDF 最高的词汇索引
   maxn_index = np.argsort( tfidf_row)[-KEYWORD_COUNT:]
   maxn_index = np.flip(maxn_index, 0)
   maxn_value = tfidf_row[maxn_index]             # 最高的 TF-IDF 值
   maxn_word = words[maxn_index]
   for i in range(KEYWORD_COUNT):
      print("%s:%0.3f, "%(maxn_word[i], maxn_value[i]), end="")
   print("")

for i in range(20):                               # 输出前 20 个邮件的关键字
   print("email %s: "%i, end="")
   get_keywords(i)
```

这段代码输出前 20 个邮件的 5 个最主要关键字，输出如图 8-3 所示。

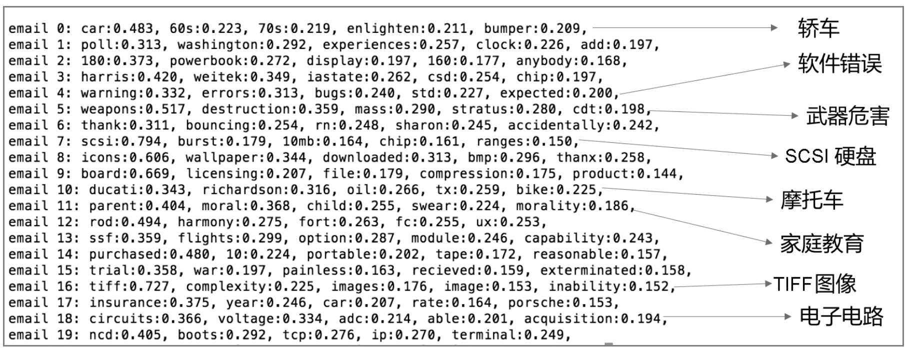


图 8-3　邮件关键字搜索

在一台普通 Mac 笔记本上这段代码的运行时间不超过一分钟。虽然这超过一万封的邮

件内容包罗万象，但可以发现即使不读邮件具体内容，仅仅根据输出的关键字就可以了解很多邮件的讨论主题。

```
print("\n".join(twenty_train.data[DOC_INDEX].split("\n")))
```

篇幅原因不再列出这些邮件的真实文本，有兴趣的读者可以通过如上代码逐个检查。

8.2　词汇处理

有了词袋模型和 TF-IDF 后，已经可以处理很多自然语言文本段落了。而本节将实践更加细节的词汇层面的两个问题：中文分词与相似词挖掘。

8.2.1　中文分词

分词是指从由字/单词组成的文本序列中提取出由一个或多个字组成的词汇，这对中文等东方语言显得尤为重要。英文等西方文字即使不进行分词处理也可以通过空格划分词汇，但汉字的词汇则灵活得多。比如句子“我来自中华人民共和国”较合理的分词方案是：“我”/“来自”/“中华人民共和国”，在后继处理中被放入词袋模型的应该是这些分词而不是单个汉字。

1. 原理

这样的分词问题几乎关乎所有汉字处理应用，从输入法的智能推荐到汉语搜索引擎，经过几十年的探索目前已经有比较成熟的分词方案。从原理上说，汉字分词可以分为基于规则匹配的方法和基于统计的方法两类。

- ◎ 前者类似于一个字典搜索系统，在分词时用最小/最大匹配、正向/逆向匹配等算法在词汇字典中找到一个句子最合适的分词方案，为了提高准确度还可以加入语法分析机制。
- ◎ 后者类似于一个机器学习方案，先分析大量的语料建立所有“字组合”出现的概率模型，在分词时把后验概率最大的分词方案作为输出结果。这方面比较典型的工具是第 6 章的隐马尔可夫模型，在 HMM 中把词汇前缀作为显示结点并把词汇后缀作为隐藏结点，这样在句子中从左往右逐个将汉字作为前缀放入 HMM 并用

Viterbi 算法预测可能的后缀是否匹配就可完成分词。

有时也可以同时使用这两种方案，比如先通过前者查找已知词组，再用概率模型匹配剩余字串。无论哪种方案都需要以庞大的语料库作为基础，因此一个好的分词方案与其说是一种算法或模型，不如说是一个系统工程。

2. 实践

由于语料库的限制，本节不再过多讨论分词的算法细节，而是直接介绍一个拿来即用的 Python 分词库——Jieba 分词。

可以使用 pip3 安装该组件：

```
# pip3 install jieba
```

在代码中引入 Jieba 后可调用 cut()函数分词：

```
>>>import jieba                                          # 引入包

>>>doc = "人工智能来源于机器人学"                          # 待分词语句
>>>seg_list = jieba.cut(doc)
>>>print("/ ".join(seg_list))
人工智能/ 来源于/ 机器人学
```

此外，Jieba 还支持所谓全模式，即在分词结果中返回所有可能的词组、允许重叠：

```
>>>seg_list = jieba.cut(doc, cut_all=True)
>>>print("Full Mode: " + "/ ".join(seg_list))
Full Mode: 人工/ 人工智能/ 智能/ 能来/ 来源/ 来源于/ 源于/ 机器/ 机器人/ 机器人学/
人学
```

显然缺省的结果较适合人类阅读，而后者可能更利于放入词袋模型进行大数据处理。此外，Jieba 还支持所谓的搜索引擎模式，可以比全模式划分出更细致的短语。Jieba 的其他特性还包括：

◎　用 load_userdict()输入自定义词典，这样可以提升特定领域的分词准确度。

◎　在 jieba.analyse 包中提供了 TF-IDF 分析等工具。

◎　支持分词后的词性（名词、动词、数词……）标注。

◎　提供分词在原文中的位置索引等。

这些特性使得我们几乎可以直接基于 Jieba 开发出关键字搜索引擎，有兴趣的读者可以参考该项目在 github.com 上的开发文档与源码。

8.2.2 Word2vec

Word2vec 是 NLP 领域近来较活跃的一种词嵌入（Word Embedding）技术，自 2013 年被谷歌开源后迅速得到广泛应用。

1. 词嵌入

所谓词嵌入就是将文本单词转换成数字或数字向量以便后续进行数字处理的方法。比如普通词袋模型中的词汇表就是一种词嵌入，因为文集中的每个单词被赋予了一个唯一的整数编号。这种词嵌入方法虽然简单易行，但是有一个很大的缺点：在单词文本与编号之间没有任何规律可循（参考 8.1.5 节中的 count_vect.vocabulary_）。

一个可能的改进措施是将 vocabulary_按照字典顺序安排索引。但这样的改进意义不大，因为字典中相邻词的词义相关性很小。

2. Word2vec 意图

Word2vec 是一种将语义相近的词编号到相近位置的技术。Word2vec 将单词映射为一个向量点，向量上的每个维度代表了单词的某种特征。如图 8-4 所示是一个对水果单词进行 Word2vec 嵌入的示意图。

	apple(苹果)	mango(芒果)	grape(葡萄)	jackfruit(菠萝蜜)
产地（温带->热带）	0.3	0.7	0.5	0.9
个体大小	0.5	0.5	0.1	0.9
口味（酸->甜）	0.5	0.6	0.7	0.8

图 8-4 对水果单词进行 Word2vec 嵌入的示意图

在如上嵌入中水果单词被映射为三维向量，比如单词 apple 被映射到向量 $\langle 0.3, 0.5, 0.5 \rangle$，这样嵌入的好处是，根据向量值就可以知道水果的不同属性，且向量距离相近的水果有一定的亲缘性。

3. CBOW 与 Skip-Gram

Word2vec 是一种无监督学习机制，依赖于海量的训练样本。其原理是分析样本中所

有单词周围的其他单词（即上下文），将具有类似上下文的单词映射为相近的向量值。根据对样本特征与标签提取的机制不同，Word2vec 有两种实现机制：Continuous Bag-of-Words（CBOW）和 Continuous Skip-Gram。前者用上下文预测中心词，而后者用中心词预测上下文，如图 8-5 所示。

Word and phrase embeddings have been **shown** to boost the performance in NLP tasks.

	特征	标签
CBOW	have, been, to, boost	shown
Skip-Gram	shown	have, been, to, boost

图 8-5　CBOW 与 Skip-Gram

图 8-5 描述的是，当上下文窗口为±2 时训练器扫描到“have been shown to boost”时 CBOW 与 Skip-Gram 不同的处理机制。CBOW 将中心词 shown 上下文中的单词作为特征，中心词 shown 被设置为它们的标签；而 Skip-Gram 则恰好与其相反。

根据谷歌 Word2vec 作者对两种方案的比较，CBOW 训练更快而 Skip-Gram 表现略好。下面用 Skip-Gram 机制讲解 Word2vec 的嵌入过程。

4. Skip-Gram 样本抽取

Word2vec 的一种实现是以单隐藏层的神经网络作为学习模型的，Skip-Gram 的训练是一个逐一扫描文本中的单词生成样本并投入神经网络的过程。如图 8-6 所示是 Skip-Gram 样本生成过程。

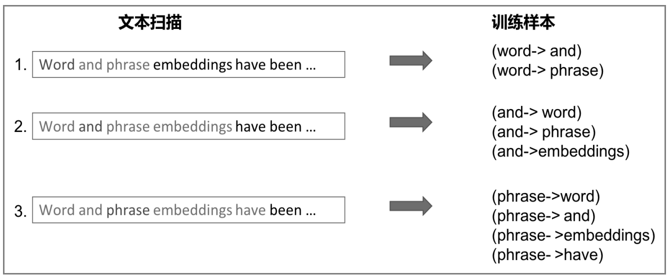


图 8-6　Skip-Gram 样本生成

该示意图的上下文窗口任然为±2，在图中第一步扫描到的单词是 word，它的上下文包括 and 和 phrase，因此在训练集生成两个样本；第二步的单词是 and，上下文包含三个单词，因此生成了三个训练样本，以此类推。与词袋模型同理，生成的样本没有考虑上下文与中心词的前后顺序，单个中心词所产生的四个样本之间完全并列。

5. 1-of-N 编码

神经网络模型的输入特征和输出标签各是一组数字向量，将样本中的这些单词放入神经网络之前需要采用 1-of-N 方案对它们进行向量化编码。

这种编码的向量维度数等于词汇表（vocabulary）中单词的个数，每个维度对应一个单词，每个单词的编码就是对应维度置 1。比如，假设词汇表按序只有四个单词：apple、mango、grape、jackfruit，则 apple 的编码为 $\langle 1,0,0,0\rangle$，mango 的编码为 $\langle 0,1,0,0\rangle$。

6. 从隐藏层提取 Word2vec

将所有样本以 1-of-N 编码放入单隐藏层的神经网络训练后，神经网络的隐藏层就成为了最终的 Word2vec 转换矩阵。关于神经网络的知识放在本书第 9 章中讨论，此处仅需理解 Word2vec 的神经网络模型近似于对如图 8-7 所示向量计算的拟合。

输入1-of-N 向量 × 隐藏层权值 → 隐藏层向量 × 输出层权值 → 输出1-of-N向量

$$\langle i_1, i_1, \ldots i_n\rangle \cdot \begin{pmatrix} w_{111} & \cdots & w_{11m} \\ \vdots & \ddots & \vdots \\ w_{1n1} & \cdots & w_{1nm} \end{pmatrix} = \langle h_1, h_2, \ldots h_m\rangle \cdot \begin{pmatrix} w_{211} & \cdots & w_{21n} \\ \vdots & \ddots & \vdots \\ w_{2m1} & \cdots & w_{2mn} \end{pmatrix} = \langle o_1, o_2, \ldots o_n\rangle$$

图 8-7　Word2vec 神经网络示意图

图中 $\boldsymbol{i}$ 向量和 $\boldsymbol{o}$ 向量分别是训练样本中的 1-of-N 单词，隐藏层向量 $\boldsymbol{h}$ 是计算的中间结果，神经网络训练的目标就是寻找到最合适的权值矩阵 $\boldsymbol{w}_1$ 和 $\boldsymbol{w}_2$ 以尽量拟合图 8-7 的等式。

注意：图 8-7 省略了神经网络中 bias、activation function 的概念，它们使神经网络具备了非线性拟合能力。

这样在神经网络的隐藏层就形成了一个从输入层获得的映射，一般情况下隐藏层向量维度远小于输入与输出层维度数（也就是单词总数），因此达到了将单词转换为数字向量的目的。此外，由于含义相近的单词（输入层）会产生类似的上下文（输出层），进而产生相近的隐藏层向量。因此，直接将每个单词放入输入层，得到的隐藏层向量就成为了 Word2vec 嵌入。

思考：如何正确地描述 Word2vec 在工业应用中能解决哪类问题？

8.2.3　实战：寻找近似词

本案例试图用 Word2vec 来挖掘“20newsgroups”文本资源库，通过它找出英文常用

词的亲缘关系。这样的挖掘并不仅仅因为有趣，在线商店可以用该方法挖掘用户评论中的名词实现评论自动归类；将文本替换为商品，超市可以用 Word2vec 挖掘客户的购买习惯。

1. Gensim 的 Word2vec 封装

由于 scikit-learn 中没有封装 Word2vec 的功能，本节使用一个专注于文本挖掘的 Python 库——Gensim 作为开发工具。像 scikit-learn 一样，Gensim 也是一个对模型进行高度封装、可以拿来即用的机器学习库。在使用之前通过如下代码进行安装。

```
# pip3 install genism
```

在代码中通过初始化一个 Word2Vec 对象即可完成一个文集的词汇向量化，比如：

```
>>>from gensim.models.word2vec import Word2Vec            # 引入对象

>>>email_data = [" ".join(email.lower().split("\n")[5: -3]).split() for
email in twenty_train.data]
>>>print(email_data[0][:10])
['i', 'was', 'wondering', 'if', 'anyone', 'out', 'there', 'could',
'enlighten', 'me']

>>>model = Word2Vec(email_data, size=100, window=10, min_count=10)
```

代码首先将“20newsgroups”中的所有文章放入 email_data 变量中，成为一个由单词组成的二维数组，然后将它作为参数初始化 Word2Vec 对象。该对象构造完成后就完成了 Word2vec 向量化，此时可以通过单词索引读取转换后的结果，代码如下。

```
>>> model['computer']
array([-2.16962433, -1.61589062,  0.53282535,  0.66242778,  0.02205923,
       0.21102917,  0.41220415,  0.58754981, -0.66076285,  0.35682362,
      …], dtype=float32)
```

如果给该索引位传入单词列表，将以二维数组方式返回多个单词的向量化结果。

2. Word2Vec 类参数

除单词列表外，Gensim 为 Word2Vec 定义了丰富的参数供调用者配置，下面列出一些比较重要的参数。

◎　sg：训练方式，0: CBOW，1: Skip-Gram。

◎ size：转换后的向量长度，即图 8-7 中隐藏层向量的长度 m。

◎ window：上下文窗口大小，即图 8-6 中上下文区域的长度。

◎ alpha、min_alpha：控制神经网络的训练速度。

◎ min_count：词汇表中单词的最少出现次数，文集中出现次数低于该值的单词将不被训练和向量化。

◎ workers：训练时启动的线程数，通常不多于当前计算机的 CPU 内核数。

3. 单词语义理解

在上述代码的 model 对象内部，实际上通过一个 KeyedVectors 类型对象 wv 来保存 Word2vec 结果。除了提供读取向量结果的能力，该对象还封装了很多基于 Word2vec 进行计算的语义函数，比如用 most_similar()函数查找近似词：

```
>>> model.wv.most_similar(positive=['two', 'three'])
[('four', 0.8489412069320679), ('five', 0.755733847618103), ('six', 0.737821638584137), ('seven', 0.6916546821594238), ('eight', 0.6915091276168823), ('thousand', 0.6770167946815491), ('hundred', 0.6712730526924133), ('hours', 0.6607673764228821), ('several', 0.6596173048019409), ('each', 0.6525782346725464)]
```

以上语句希望找出单词 two、three 近似的词。虽然没有为模型输入过任何英文语法知识，但是 Word2vec 竟然精确地给出了 four、five、six…这些英文数词。每个单词后跟的数字表示匹配程度，该值来自目标单词向量与匹配单词之间的 cosine-similarity 距离（关于 cosine-similarity 可以参考第 4 章）。

还可以用 doesnt_match()函数找出若干单词中的最“与众不同”者：

```
>>> model.wv.doesnt_match("computer pc disk fish video".split())
'fish'
```

模型非常聪明地从若干计算机术语中找到了“fish”这一异类单词。用 similarity()函数查询任意两个单词的距离，数值越大说明越接近：

```
>>> model.wv.similarity('leader', 'king')
0.77802590283567785
>>> model.wv.similarity('leader', 'man')
0.53062029986626924
>>> model.wv.similarity('leader', 'light')
```

```
0.047387288597156296
```

上述代码分别用三个单词比较与 leader 的距离，从结果数据来看，与 leader（领导者）距离从近到远的是：king（国王）、man（普通人）、light（灯光），这也相当符合语义逻辑。

4. 语义聚类可视化

第 5 章中曾经提到过降维模型 t-SNE 的研究初衷是实现高维数据的可视化，这里尝试用该模型绘制出所有“20newsgroups”中最频繁出现的单词之间的亲缘关系图，该过程可以通过如下步骤完成。

- ◎ 从“20newsgroups”中提取文集转换为单词二维数组，并去除标点符号等非单词符号。
- ◎ 用 Gensim 将单词 Word2vec 向量化。向量长度可以任取，通过 min_count 设置只读取出现次数大于 2000 的单词。
- ◎ 用 t-SNE 将单词向量降到二维，以便在平面上绘制。
- ◎ 用 Matplot 绘制降维结果。

具体代码如下。

```
########### 读取文集 #############
from sklearn.datasets import fetch_20newsgroups

twenty_train = fetch_20newsgroups(subset='train')
sentences = []
for doc in email_data:                          # 删除非英文或数字字符
    sentence = [ word for word in doc if word.isdigit() or word.isalpha()]
    sentences.append(sentence)

########### Word2vec 向量化 ###############
from gensim.models.word2vec import Word2Vec

model = Word2Vec(sentences, size=100, window=10, min_count=2000, workers=4)
vocab = list(model.wv.vocab)
X = model[vocab]

###########  t-SNE 降维 ################
from sklearn.manifold import TSNE
```

```
tsne = TSNE(n_components=2)
X_tsne = tsne.fit_transform(X)

########### Matplot 绘制 ##################
import matplotlib.pyplot as plt
from adjustText import adjust_text

fig, ax = plt.subplots()
plt.plot(X_tsne[:, 0], X_tsne[:, 1], 'bo')
texts = []
for x, y, s in zip(X_tsne[:, 0], X_tsne[:, 1], vocab):
    texts.append(plt.text(x, y, s))
adjust_text(texts,arrowprops=dict(arrowstyle='->', color='red'))
plt.show()
```

代码执行结果如图 8-8 所示。

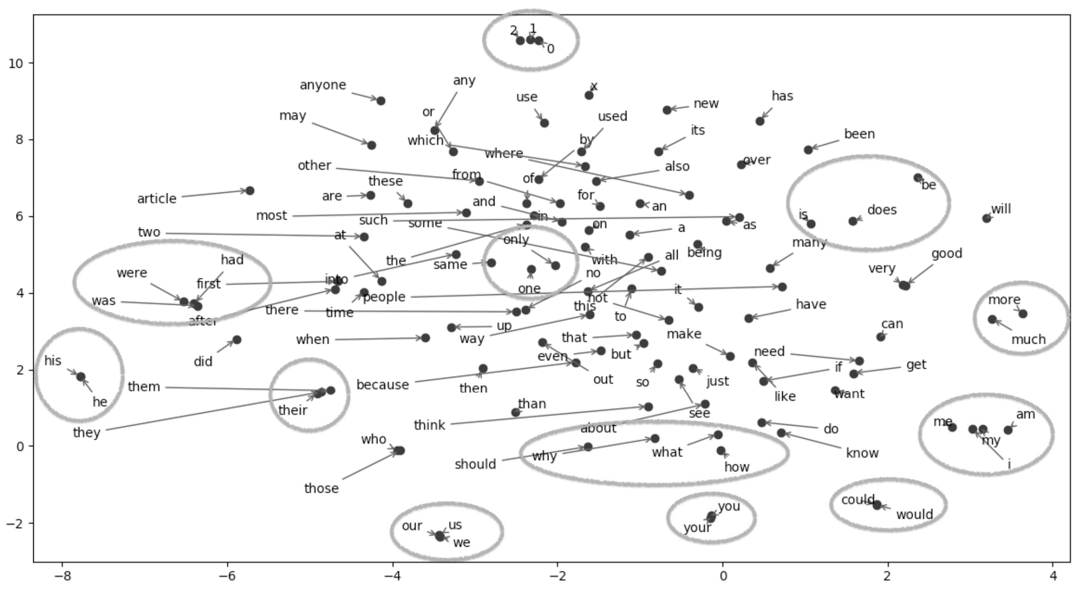


图 8-8　语义聚类可视化代码执行结果

图中的点、线、文字为程序绘制，椭圆区域是笔者手动圈出的。可以发现模型自动识别出了很多相关语义组。

◎　第一人称代词 I、me、my，以及它们的复数 we、us、our。

◎　第三人称代词 he、his，以及它们的复数 they、them、their。

◎　数词：0、1、2，其他数词未被识别是因为出现次数没有达到 2000 次的要求。

◎　助动词 be、is、does，以及它们的过去式 was、been、had。

◎　疑问助词 why、what、how 等。

这样的英文单词分类能力似乎已经达到了人类学习英文一至两年的水平。由于二维空间的限制，无法再图形化展示出更多非常用词的聚类效果。在特定领域的实际应用中，完全可以通过缩小词汇整体的范围进行更小范围的语义挖掘。

5. Word2vec 不足的思考

以基于上下文数据的训练作为出发点，本节已经展示了 Word2vec 的强大力量。虽然该模型能自动挖掘单词/短语之间的近似程度，但也会把一些语境相似、语义相反的词归并到一起，比如 good 和 bad、fast 和 slow，这是因为它们出现的上下文确实非常相似：

◎　Jame is a good student, he is 10 years old.

◎　Jame is a bad student, he is 10 years old.

这种情况对于形容词来说非常普遍，导致 Word2vec 无法区分它们。因此在类似于网店评论自动归类这类应用中，对于这类形容词还需要加入有监督的分类处理。

8.3　主题模型

在自然语言中，每篇文章或对话背后都由一个或若干主题构成，这些主题可大可小，从军事、政治、经济，到番茄炒蛋的制作、模型飞机的操控技巧等。主题模型（Topic Model）是在自然语言处理中比较活跃的领域，它的目标是从一个文集中用机器学习或统计方法挖掘出每个文档所蕴含的主题。本节从通用的三层主题模型表示方法开始，逐步介绍具体的算法 NMF、LSA、PLSA 及概率模型 LDA（Latent Dirichlet Allocation）。

8.3.1　三层模型

机器学习中主题模型的具体算法有很多，从纯频率统计的 NMF、LSA 到概率性的

LDA。但万变不离其宗，所有算法都有一个共同的目标，就是建立一个“文档→主题→单词”的三层模型。在这个三层模型中，假设每篇文档/文章由若干主题组成、而每个主题由若干单词组成。三层模型用量化的形式表达了它们之间的如下关系。

◎ 每个文档与各主题的相关程度。

◎ 每个主题与各单词的相关程度。

比如在一篇新闻联播的报道中，很可能包括主题“内政”“工业”“外交”等。而每个主题是由关键词构成的，比如工业主题中的关键词很可能有“钢产量”“GDP”等。

所有具体算法的输入都是词袋模型的“文档→单词”关系（当然可以是 TF-IDF），而目标就是从该矩阵中提取出如上两种关系，如图 8-9 所示。

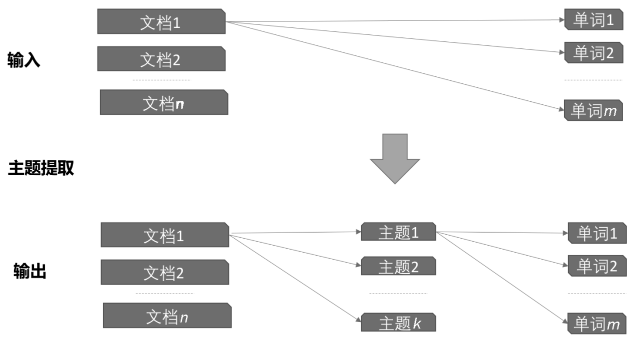


图 8-9　三层模型

不同算法对提取出的三层模型表达方式有所区别，比如 NMF 用两个矩阵分别表达两种相关性、LSA 用三个矩阵的两两相乘表达，而 LDA 用两组 Dirichlet 分步表达。

8.3.2　非负矩阵分解

非负矩阵分解（Non-negative Matrix Factorization，NMF）是一种将一个矩阵近似转换为两个矩阵相乘形式的想法：

$$\boldsymbol{A}(u\times v)\approx \boldsymbol{W}(u\times k)\boldsymbol{H}(k\times v)$$

其中数据矩阵 $\boldsymbol{A}$ 是 $u\times v$ 形状的非负矩阵，输出 $\boldsymbol{W}$ 和 $\boldsymbol{H}$ 是 $u\times k$ 与 $k\times v$ 形状的非负矩阵，由于 k 可以远小于 u 与 v，因此转换后降低了数据总量，达到了降维和数据压缩的效果。

该想法由来已久，但是 $\boldsymbol{W}$ 与 $\boldsymbol{H}$ 的求解并不容易，是一个带约束的优化问题。直到 1999

年两位科学家 D.D.Lee 和 H.S.Seung 在 *Nature* 杂志中发布了用 EM 算法寻找近似解的倍增更新法（Multiplicative Update Rule），才使得 NMF 逐渐流行起来。

1. 在主题模型中的应用

主题模型是NMF在机器学习中的主要应用之一，这时矩阵 $\boldsymbol{A}(u\times v)$ 就是词袋模型矩阵，其中 u 是文档总量，v 是单词总量，分解后的形式如图 8-10 所示。

$$\text{文档}\left\{\overbrace{\begin{pmatrix} A_{11} & \cdots & A_{1v} \\ \vdots & \ddots & \vdots \\ A_{u1} & \cdots & A_{uv} \end{pmatrix}}^{\text{单词}}\right. \rightarrow \text{文档}\left\{\overbrace{\begin{pmatrix} W_{11} & \cdots & W_{1k} \\ \vdots & \ddots & \vdots \\ W_{u1} & \cdots & W_{uk} \end{pmatrix}}^{\text{主题}}\right. \left.\overbrace{\begin{pmatrix} H_{11} & \cdots & H_{1v} \\ \vdots & \ddots & \vdots \\ H_{k1} & \cdots & H_{kv} \end{pmatrix}}^{\text{单词}}\right\}\text{主题}$$

图 8-10　NMF 主题模型

NMF 分解后，$\boldsymbol{W}$ 矩阵表示的就是“文档→主题”关系，而 $\boldsymbol{H}$ 矩阵则是“主题→单词”关系，$\boldsymbol{W}$ 中单元格数值越大表明该单词与相应主题的关系越密切，矩阵 $\boldsymbol{H}$ 同理。

2. k 值的选取

与 K-means 等聚类算法需要调用者设置 k 值一样，NMF 中的 k 值也需要模型应用者定义，而该 k 值的选取并没有统一的标准。由于 NMF 是一种近似分解，分解后的 $\boldsymbol{WH}$ 矩阵相比原始矩阵 $\boldsymbol{A}$ 会丢失一些信息，所以一般来说原始矩阵 $\boldsymbol{A}$ 的数据越复杂则需要设置相对高的 k 值，使得经过转换后丢失的信息更少。

3. NMF 的其他应用

NMF 作为一种矩阵分解法不仅应用在 NLP 中，还在图像分类、商品推荐等领域得到应用。对于图像识别来说，$\boldsymbol{A}$ 矩阵中的每一列是一幅图像上的所有像素值，而提取到的“主题”就是识别出的图像内容。对于商品推荐来说，$\boldsymbol{A}$ 矩阵的每一列是一个用户采购过的所有商品的记录，而提取到的“主题”则是用户的购买习惯。即在 $\boldsymbol{H}$ 矩阵中可以获得每个用户有哪些购买习惯，而从 $\boldsymbol{W}$ 矩阵中可以获得该购买习惯的用户倾向于购买哪些商品。

8.3.3　潜在语意分析

潜在语意分析（Latent Semantic Analysis，LSA）是一种最传统的主题模型，早在 1988 年就被发明用于信息检索，当时该技术被称为 latent semantic indexing（LSI），在 2005 年

该技术被用于语音识别后被称为 LSA。由于 LSI/LSA 是矩阵奇异值分解（SVD）的最早应用之一，在 NLP 领域 SVD 有时也被用于指代 LSI/LSA。

1. SVD 分解计算

与 NMF 类似，LSA 也是一种矩阵分解方法，它首先对词袋模型矩阵进行 SVD 分解：

$$\boldsymbol{A}(u\times v)=\boldsymbol{U}(u\times u)\boldsymbol{\Sigma}(u\times v)\boldsymbol{V}(v\times v)$$

其中 $\boldsymbol{\Sigma}$ 是非对角线元素都为零的对角阵，对角线上的值称为 $\boldsymbol{A}$ 的奇异值，该分解可以通过如下步骤完成：

- ◎ 对方阵 $\boldsymbol{AA}^{\mathrm{T}}$ 进行特征值分解，特征向量组成 $\boldsymbol{U}$。
- ◎ 对方阵 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$ 进行特征值分解，特征向量组成 $\boldsymbol{V}$。
- ◎ $\boldsymbol{AA}^{\mathrm{T}}$ 与 $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}$ 的特征值必定相等，任取它们之一的特征值并开平方根构成 $\boldsymbol{\Sigma}$ 。

2. 主题提取

从计算过程可知，$\boldsymbol{A}$ 矩阵的奇异值就是 $\boldsymbol{AA}^{\mathrm{T}}$ 的特征值，在本书第 5 章中经常用到的提取最大特征值/特征向量以实现降维的方法也适用于奇异值。LSA 正是利用这种 SVD 降维达到主题提取目的：

- ◎ 矩阵 $\boldsymbol{U}$ 中的每一列对应一个奇异值，矩阵 $\boldsymbol{V}$ 中的每一行对应一个奇异值。
- ◎ 在 $\boldsymbol{\Sigma}$ 中奇异值从大到小降序排列，左上角的 k 个奇异值是最大的 k 个奇异值，取这些奇异值对应的 $\boldsymbol{U}$ 中的列与 $\boldsymbol{V}$ 中的行可以组成降维后的 SVD 分解：

$$\boldsymbol{A}(u\times v)\approx\boldsymbol{U}(u\times k)\boldsymbol{\Sigma}(k\times k)\boldsymbol{V}(k\times v)$$

将主题看成单词的降维结果，则 k 也就对应了挖掘出的 k 个主题。一般情况下这种 SVD 降维只需保存很少的奇异值就可保留大部分的原矩阵信息，因此 k 可以很好地表征原矩阵中的主题。

3. 相关性提取

如果原始矩阵中 u 是文档总量并且 v 是单词总量，则转换后矩阵 $\boldsymbol{U}(u\times k)$ 理解为文档降维矩阵，$\boldsymbol{\Sigma}(k\times k)$ 是主题矩阵，$\boldsymbol{V}(k\times v)$ 是单词降维矩阵，它们的乘积就是相关性：

◎　$U(u\times k)\Sigma(k\times k)$ 生成的 $u\times k$ 矩阵是 u 个文档与 k 个主题之间的相关程度。

◎　$\Sigma(k\times k)V(k\times v)$ 生成的 $k\times v$ 矩阵是 k 个主题与 v 个单词之间的相关程度。

8.3.4　隐含狄利克雷分配

隐含狄利克雷分配（Latent Dirichlet Allocation，LDA）是用概率论建模的主题模型，自 2003 年由 David Blei.、吴恩达等人提出后，目前仍是谷歌等前沿公司的主题模型工具。

注意：LDA 其实是贝叶斯网络的一种应用，本节以第 7 章知识作为基础。

与 NMF、LSA 等用矩阵分解进行主题挖掘的模型不同，LDA 是一种利用共轭先验建模的概率模型。前者常被称为频率派模型，而后者是贝叶斯派。LDA 在某种意义上是对另一种频率模型 pLSA（probabilistic Latent Semantic Analysis）的改进，因此这里从 pLSA 说起。

1. pLSA 模型

从名称可以看出，pLSA 与 LSA 有密切的关系，它们之间的区别在于 LSA 使用矩阵分解挖掘主题，而 pLSA 通过对概率分布进行参数估计挖掘主题。

pLSA 假设每个文档与主题之间、主题与单词之间都存在着一个多项式分布（Multinomial Distribution），而文档是按这两类多项式分布进行层叠采样的结果，如图 8-11 所示。

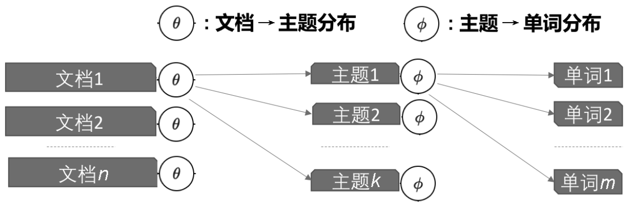


图 8-11　pLSA 模型

图中每个文档、主题都有一个自己的多项式分布，如果确定了主题数量 k，则可由训练样本通过参数估计方法计算出图中每个多项式分布的参数 $p_1,p_2\ldots p_j$，这些概率参数也就说明了文档与主题、主题与单词之间的相关程度。

2. LDA 模型

在贝叶斯学派的科学家看来，pLSA 完全根据样本来推算两类多项式分布，无法在训练前加入先验知识。如果样本数量有限或者分布不均衡，pLSA 的结果与真实值的差距可能会很大。

由于 pLSA 本身使用多项式分布建模，使得在模型中加入它的共轭分布——狄利克雷（Dirichlet）分布后就可以顺理成章地转变为一种贝叶斯派模型：用先验协助参数估计，并且生成后验以备后续使用。在 pLSA 这样贝叶斯化之后，也就成为了所谓的 LDA 模型，如图 8-12 所示。

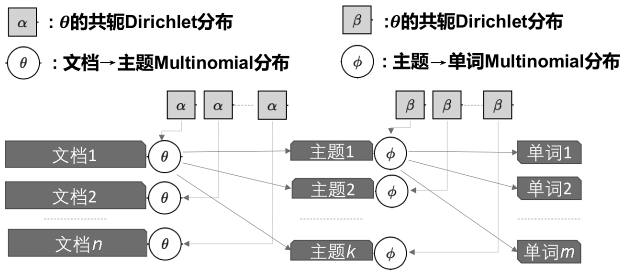


图 8-12　LDA 模型

图 8-12 形成了一个庞大的贝叶斯网络，在其中 θ 与 ϕ 还是 pLSA 中待参数估计的多项式分布，但是在每个 θ 与 ϕ 之上又增加了各自的共轭分布 α 与 β，这些共轭分布在训练之前可以作为超参数传入 LDA 模型中（先验作用），在训练之后则保存了根据样本调整后的数据（后验作用）。

技巧：现在可以知道，划分机器学习频率派模型和贝叶斯派模型的关键不在于模型中是否使用了概率分布；而在于模型是否可以输入先验，并且在训练后得到后验。

3. LDA 的训练

LDA 的训练是指从语料词袋矩阵学习到图 8-12 中所有参数的过程。由于文档的数量 n 和单词的数量 m 都可以从词袋矩阵中获得，因此在定义了主题数量 k 后就完全确定了图 8-12 所示贝叶斯网络的网络结构。这样 LDA 的训练成为了贝叶斯网络的推理过程。

如果将图 8-12 看成从左到右横卧的一棵树，则 LDA 网络实际是一种以单词的出现频率作为叶子结点的树形结构。还记得贝叶斯网络只要观测到（observe）任何多个结点状态就可以推理出网络中其他参数的性质吗？这就是 LDA 训练的关键了：将词袋模型作为观

测结点输入 LDA 网络，然后推理出网络中的所有 θ 与 ϕ 参数并更新 α 与 β 分布参数！

理论上这个训练过程可以使用任何贝叶斯网络推理算法，但由于网络非常庞大（有 $n+m$ 个 Multinomial 和 Dirichlet 分布），所以实际应用中只能选择 VB 或 MCMC 这样的近似推理算法，具体过程可回顾本书第 7 章。

8.3.5　实战：使用工具包

如上所说的主题模型都绝非只能空谈，在 Scikit-learn 中已经有对它们的实现，所有主题模型都封装在 sklearn.decomposition 包中。

1. NMF 类

非负矩阵分解类直接以 NMF 命名，示例代码如下：

```
>>>import numpy as np
>>>X = np.array([[1, 1, 3], [2, 1, 2], [3, 1.2, 1], [4, 1, 2],
                  [5, 0.8, 0.3], [6, 1, 3.5]]) # 输入矩阵
>>>from sklearn.decomposition import NMF

>>>model = NMF(n_components=2)                    # 设置 k=2
>>>W = model.fit_transform(X)                     # 分解
>>>H = model.components_                          # 查询 H 矩阵

>>>print(X)                              # 原始输入矩阵，形状为 6×3
[[ 1.   1.   3. ]
 [ 2.   1.   2. ]
 [ 3.   1.2  1. ]
 [ 4.   1.   2. ]
 [ 5.   0.8  0.3]
 [ 6.   1.   3.5]]
>>>print(W)                              # 分解后的 W 矩阵，形状为 6×2
[[ 0.33084809  1.19307325]
 [ 0.65962477  0.77511704]
 [ 0.99581791  0.37426809]
 [ 1.29967271  0.7019512 ]
 [ 1.62715202  0.        ]
 [ 1.92659157  1.2008453 ]]
>>>print(H)                              # 分解后的 H 矩阵，形状为 2×3
```

```
[[ 3.08109887  0.42420192  0.20037681]
 [ 0.          0.58775875  2.49031343]]
```

上述代码中，用 k=2 对一个 6×3 的输入矩阵进行 NMF 分解，从.fit_transform()函数的返回值获得 $\boldsymbol{W}$ 矩阵，其后通过 components_属性读取 $\boldsymbol{H}$ 矩阵。

在 NMF 类对象初始化时，除了可以用 n_components 设置 k 值，还有一些参数用于控制分解算法的种类与参数，比如 init 参数设置迭代的初始值，solver 参数设置使用梯度下降算法还是倍增更新算法，max_iter 参数设置最大迭代次数等。

2. TruncatedSVD 类

TruncatedSVD 是一个 SVD 降维模型，如果用文本词袋模型作为输入，那么它就是本节提及的 LSA/LSI 模型。其使用方法与 NMF 类似，从 fit_transform()函数返回值提取分解后的左半部分矩阵 $\boldsymbol{U}(u\times k)\boldsymbol{\Sigma}(k\times k)$，然后从 components_属性读取右半部分矩阵 $\boldsymbol{\Sigma}(k\times k)\boldsymbol{V}(k\times v)$，比如：

```
>>>from sklearn.feature_extraction.text import CountVectorizer
>>>from sklearn.decomposition import TruncatedSVD

>>>count_vect = CountVectorizer(stop_words='english')  # 词袋模型
>>>X_counts = count_vect.fit_transform(["I love machine learning",# 文集
                              "I love python",
                              "The weather is good"])
>>>lsa = TruncatedSVD(n_components=2)                  # LSA/SVD 对象

>>>doc_topic = lsa.fit_transform(X_counts)             # SVD 分解
>>>topic_word = lsa.components_

>>>print("shape of bag of words: ", X_counts.shape)
>>>print("shape of topic_word is ", topic_word.shape)  # 主题→单词 矩阵
>>>print("doc_topic:\n", doc_topic)                    # 文档→主题 矩阵
```

上述代码的执行结果为：

```
shape of bag of words:  (3, 6)
shape of topic_word is  (2, 6)
doc_topic:
 [[  1.61803399e+00   0.00000000e+00]
 [  1.00000000e+00   3.14439865e-16]
 [ -2.22342561e-16   1.41421356e+00]]
```

这里单词总数是 6，小于文档中的词汇总量 9，这是因为在词袋模型中配置了停止词。比较文档->主题矩阵中的数值大小，文档 1 与文档 2 属于主题 1，而文档 3 属于主题 2。

3. LatentDirichletAllocation 类

LatentDirichletAllocation 提供了 LDA 模型封装，其使用方法几乎与 TruncatedSVD 完全一样，代码如下。

```
>>>from sklearn.decomposition import LatentDirichletAllocation

>>>lda = LatentDirichletAllocation(n_components=2, doc_topic_prior=1)
>>>doc_topic = lda.fit_transform(X_counts)
>>>topic_word = lda.components_
```

上述代码的执行结果留给读者自行尝试。其与 TruncatedSVD 的区别主要在于一些特有的模型参数：doc_topic_prior 和 topic_word_prior 用于设置两类狄利克雷分布的先验参数，learning_method 用于设置是否启用增量学习。

有些遗憾的是，scikit-learn 对狄利克雷先验的配置功能非常有限，即使用均匀的先验分布，doc_topic_prior 与 topic_word_prior 也只能配置单一的 float 数值。

8.4　实战：用 LDA 分析新闻库

本节介绍用 scikit-learn 的 LDA 工具分析“20newsgroups”文本语料库，并利用 perplexity 评估分析结果。

8.4.1　文本预处理

现实世界中的文本总是包含了诸多内容的，有些是有助于语义分析的，而有些是无用的，如图 8-13 所示是“20newsgroups”语料库中的第一条新闻。

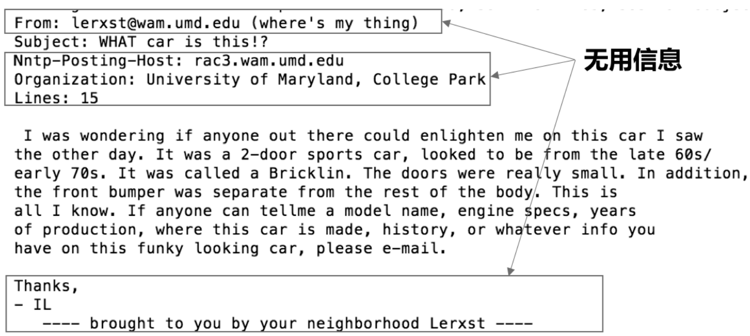


图 8-13 “20newsgroups”语料库中的第一条新闻

图中的邮件头、正文行数、致谢等信息显然与该条新闻的实际内容无关，在机器学习之前剔除这些明显无用的噪声信息有助于提高后续处理的有效性。只要按照文本内容编写一些简单规则就可以达到该目的，比如：

```
def filt_line(email):
    lines = []
    for line in email.split("\n"):
        line = line.strip()
        if len(line) < 20 or line.startswith("-") or \
           line.find("Subject:")<0 and line.split()[0][-1]==":":
            continue
        lines.append(line)
    return "\n".join(lines)
```

上述代码可以过滤掉过短、以“-”或“xx:”开头的段落。在实际应用中，这部分处理需要根据文本媒介格式自行定义。

此外，语义分析通常关注的是名词、形容词、动词，对 I、this、3 这样的代词、数词等则关注较少，可以用 Python 库的 NLTK 对词性进行过滤：

```
import nltk
from nltk.stem.porter import PorterStemmer
def filt_word(email):
    email = email.lower()
    lines = []
    ps = PorterStemmer()                                # 词干提取对象

    for line in email.split("\n"):
        words = nltk.word_tokenize(line) # 分词
```

```
        words = [w for w in words if w.isalpha()]   # 仅保留英文单词
        taged_words =nltk.pos_tag(words)            # 获取词性标签
        words = [w for w, pos in taged_words if pos in ['NN',]  # 仅保留名词
        words = [ps.stem(w) for w in words]         # 词干化
        lines.append(" ".join(words))
    return "\n".join(lines)
```

NLTK 仍然可以通过 pip3 install 安装，如上代码主要使用了 NLTK 提供的两种功能：

◎　用 NLTK 查询所有单词的词性，并只保留名词（‘NN’）。

◎　将所有单词词干化，比如将复数名词 bags 转换为 bag。

8.4.2　训练与显示

现在可以将预处理后的“20newsgroups”文本资源加载到 LatentDirichletAllocation 模型中，该新闻库与 LDA 模型的基本使用前文已经描述，直接列出相关代码：

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import LatentDirichletAllocation

twenty_train = fetch_20newsgroups(subset='train')
email_data = twenty_train.data[:30]                     # 仅取前 30 条新闻
original_email = email_data

email_data = [filt_line(email) for email in email_data]    # 行过滤
email_data = [filt_word(email) for email in email_data]    # 单词过滤

count_vect = CountVectorizer(stop_words='english')
X_counts = count_vect.fit_transform(email_data)            # 建立词袋
lda    =    LatentDirichletAllocation(n_components=5,    random_state=0,
learning_method=’batch’)

doc_topic = lda.fit_transform(X_counts)        # 训练LDA模型、获得doc_topic
topic_word = lda.components_                   # 获得 topic_word
```

出于降低实践时程序运行时间的考虑，上述代码仅引用语料库中的前 50 条新闻。运行上述代码后就获得了两类关系矩阵 doc_topic 和 topic_word，查看它们的内容：

```
>>>print(doc_topic.shape, topic_word.shape)
(50, 5) (5, 1170)
```

```
>>> doc_topic[0]
[ 0.97948506  0.00512866  0.00512902  0.00512885  0.00512841]
```

由 doc_topic 与 topic_word 的形状可知，当前 LDA 模型训练了 30 篇文章、10 个主题、1170 个单词。进一步查看矩阵内容时会发现数值杂乱难以阅读，此时可编写工具函数辅助结果显示：

```
import numpy as np
dict_word = {count_vect.vocabulary_[key]: key for key in count_vect.
vocabulary_.keys()}
# 单词索引->文本映射
words = np.array([dict_word[idx] for idx in range(len(dict_word))])

def print_top_words(words, topic_word, topic_idx, n=10):
    # 查找主题中权值最大的 n 个单词索引
    maxn_index = np.argsort(topic_word[topic_idx])[-n:]
    maxn_index = np.flip(maxn_index, 0)
    maxn_value = topic_word[topic_idx][maxn_index] # 记录单词权值
    maxn_word = words[maxn_index]                  # 记录单词文本

    print("topic %s: %s"%(topic_idx, ", ".join(["%0.3f*%s"%(value, word) for
value, word in zip(maxn_value, maxn_word)])))

def print_doc_topics(words, doc_topic, topic_word, doc_idx, n=3):
    # 查找文档中权值最大的 n 个主题索引
    maxn_index = np.argsort(doc_topic[doc_idx])[-n:]
    maxn_index = np.flip(maxn_index, 0)
    maxn_value = doc_topic[doc_idx][maxn_index]    # 记录主题权值

    print("doc %0.3f:"%doc_idx)
    for i in range(n):
        print("   weight: %s "%(maxn_value[i], ), end='')
        print_top_words(words, topic_word, maxn_index[i])
```

由于 LDA 中使用的都是单词索引，代码中首先从词袋的 vocabulary_属性建立了“单词索引→单词文本”映射变量 words 以备后续查表。然后分别定义了格式化输出两类关系的函数 print_top_words 和 print_doc_topics。此时可以较友好地查询到 LDA 训练结果，代码如下。

```
>>>print_doc_topics(words, doc_topic, topic_word, 0)
doc 0:
  weight: 0.959 topic 1: 11.202*option, 11.200*articl, 10.203*control,
```

```
9.200*tiff, 7.203*scsi, 7.203*anyon, 7.202*chip, 7.201*power, 7.200*time,
7.197*car
      weight: 0.010 topic 0: 24.200*insur, 11.210*car, 11.204*year,
9.201*articl, 9.200*board, 8.200*rate, 8.200*mass, 7.200*compani,
7.200*destruct, 7.200*catalog
      weight: 0.010 topic 4: 5.207*car, 5.200*stereo, 4.200*cover, 4.200*radio,
4.200*face, 4.200*teenag, 3.202*treatment, 3.202*someth, 3.200*case,
3.200*theft

    >>>print(original_email[0])
    I was wondering if anyone out there could enlighten me on this car I saw
    the other day. It was a 2-door sports car, looked to be from the late 60s/
    early 70s. It was called a Bricklin. The doors were really small. In addition,
    the front bumper was separate from the rest of the body. This is
    all I know. If anyone can tellme a model name, engine specs, years
    of production, where this car is made, history, or whatever info you
    have on this funky looking car, please e-mail.
```

上述代码打印了语料库中第一个文件的前 3 个最重要的主题和每个主题中最重要的 10 个单词。由于第一个主题的权值就达到了 0.979，这意味着该份文档只有一个重要主题。从后面打印的文档原文来看，该份文档是谈论关于 car 的事情，但是关键词 car 在识别出的唯一主题中仅仅是第 10 重要的关键词。很明显该 LDA 模型对文档的拟合还不够好，需要调参以提高拟合程度。

8.4.3　困惑度调参

就像分类/回归问题中可以用精确度、召回率等指标衡量一个模型的好坏一样，主题模型的一个重要指标是困惑度（Perplexity）。它的基本思想是用训练好的模型计算测试集中文档的生成概率，测试集文档被产生的概率越高则说明模型越好。困惑度计算的原理非常简单，对 LDA 来说：

$$\mathcal{L}(w)=\sum_{d}\log P(w_d\mid\phi,\alpha)$$

其中 w 是测试集总体，w_d 是单个测试文档，ϕ 和 α 是训练后模型已知参数（图 8-12 中的主题多项式分布和文档狄利克雷分布），因此 $\mathcal{L}(w)$ 就是一个从已知贝叶斯网络计算条件概率的问题。

在 LatentDirichletAllocation 类对象中提供了计算困惑度的方法 perplexity()，该参数也被 scikit-learn 中的交叉验证调参模块用于评估模型的有效程度。因此可以编写如下代码搜

索 LDA 的最佳超参数：

```
from sklearn.model_selection import GridSearchCV
parameters = {'n_components':range(5, 11, 5),          # 搜索参数空间
            'doc_topic_prior':(0.001, 0.01, 0.1, 0.5, 1),
            'topic_word_prior':(0.001, 0.01, 0.1, 0.5, 1),
            'learning_method':('batch',),
            'random_state':(0, )}

lda = LatentDirichletAllocation()
model = GridSearchCV(lda, parameters, cv=2)            # 交叉验证搜索对象
model.fit(X_counts)                                    # 训练并测试

print(model.cv_results_)
```

GridSearchCV 对象的 fit()函数会尝试 parameters 中定义参数的所有组合形式（本例中为 2×5×5×1×1=50 种组合），分别对每一种组合执行交叉验证，将结果保存在 cv_results_ 属性中。训练后，cv_results_是一个庞大的如下形式的字典：

```
{
'mean_fit_time': array([ 0.04911995,  0.04826891,…     # 平均训练时间
'params': [{'doc_topic_prior': 0.01, 'learning…        # 参数组合
'mean_test_score': array([-87000.35820776, -16…        # 平均得分（困惑度）
'rank_test_score': array([27, 15,  1, 26,…             # 分数排行
…
}
```

其中每一个键是一个评估指标，所有的值都是一个有相同长度的列表/数组（本例中长度为 50）。所有值列表中的元素一一对应，比如下列代码可以找到最佳分数排行的参数组合：

```
best_idx = np.argmin(model.cv_results_['rank_test_score'])
print("Best params is: ", model.cv_results_['params'][best_idx])
```

输出是：

```
Best params is:  {'doc_topic_prior': 1, 'learning_method': 'batch',
'n_components': 5, 'n_jobs': 1, 'random_state': 0, 'topic_word_prior': 1}
```

至此获得了较好的模型训练参数，用该组参数训练 LDA 并查询第 1 条新闻的主题结果得到：

```
doc 0:
  weight: 0.768 topic 0: 24.984*insur, 19.903*car, 12.588*year,
```

```
10.170*articl,  9.999*board,  8.993*rate,  8.971*mass,  8.067*compani,
7.968*destruct, 7.951*catalog
      weight: 0.064 topic 1: 14.166*articl, 12.369*option, 12.256*control,
9.950*tiff, 8.978*parent, 8.960*god, 8.950*space, 8.780*scsi, 8.277*chip,
8.059*power
      weight:  0.059  topic  4:  5.969*stereo,  5.936*treatment,  5.916*car,
4.990*someth, 4.971*teenag, 4.970*face, 4.970*cover, 4.970*radio, 4.941*mask,
3.972*son
```

该文档第 1 个主题的第 2 个关键词是“car”，后面的“year”“board”“company”等都是该语料中作者提出的问题，可见该主题确实更符合文本的实际情况。

本案例为了尽量减少 LDA 模型训练和自动调参的运行时间，使用了非常少的文档进行分析，在实际应用当中完全可以将远大于此的语料库放入 LDA 模型中。

此外，前文中提到过目前 scikit-learn 的 LDA 工具有一个缺陷是无法以列表形式配置 Dirichlet 先验，使得调用者无法输入非均匀的先验分布；而 Gensim 中对该功能有较好支持，但 Gensim 中缺少完善的自动调参工具，在实际应用中可以综合两种工具一起使用。

8.5　本章内容回顾

- ◎ 当前工业界聊天机器人并非有人类智慧，它背后是一个领域内的文本分类器。
- ◎ 词袋模型是当前将文本转换为数字向量输入机器学习模型中的主要方法。
- ◎ TF-IDF 技术可以无监督地找出文章中的关键字，但是需要将所有语料一次性放入计算，无法实现增量学习。
- ◎ 分词是中文 NLP 的必备步骤，其原理主要是基于字典搜索和概率模型预测，Jieba 是 Python 中较成熟的中文分词工具。
- ◎ Word2Vec 用于寻找文本中的近似词，其背后是基于对所有单词上下文的分析。
- ◎ 主题模型是指一类能挖掘出“文档→主题→单词”三层模型关系的 NLP 工具，其中 NMF、LSA/LSI/SVD、pLSA 是频率派主题模型，LDA 是贝叶斯派主题模型。
- ◎ 划分机器学习频率派模型和贝叶斯派模型的关键不在于模型中是否使用了概率分布，而在于模型是否可以输入先验知识，并且在训练后得到后验知识。

◎ NMF 将词袋矩阵直接分解为两个较小矩阵，分别表示“文档→主题”关系和“主题→单词”关系。

◎ LSA/LSI 的本质是 SVD 降维。

◎ LDA 是贝叶斯网络在主题模型上的应用，它使用 Multinomial 和 Dirichlet 共轭分布对所有文档与主题进行建模。

◎ LDA 的训练过程就是贝叶斯网络的推理过程，达到计算出网络中未知模型参数的目的。

◎ scikit-learn、Gensim、NLTK 中有易于使用的词袋模型、Word2vec、主题模型工具包，实践中可权衡它们的优势综合运用。

9

第 9 章 深度学习

理论上深度学习是指一类用复杂多重非线性结构分析数据的抽象算法，现在人们谈论的深度学习多特指以深度神经网络作为工具的机器学习模型，当前被广泛应用的卷积神经网络（CNN）、递归神经网络（RNN）等均属这一范畴。本章以深度学习工具 TensorFlow 为依托，深入浅出地介绍了神经网络基本原理、CNN 与 RNN 的理论与应用，主要内容如下。

- ◎ 神经网络基础：动机、网络结构、反向传播算法与梯度消失问题。
- ◎ TensorFlow 应用：学习 TensorFlow 中张量、评估器、图、会话等基本要素，具备用其进行深度学习开发的能力。
- ◎ 卷积神经网络（CNN）：通过卷积层、池化层等技术建立较深的神经网络。
- ◎ 网络优化策略：规范化、剪枝、多种优化算法比较。
- ◎ 循环神经网络（RNN）：时间循环神经网络，LSTM 原理，空间循环（递归）神经网络。

◎ 前沿精选：物件检测模型、密连接网络、胶囊网络等。

◎ 案例：CNN 与 RNN 在图像识别与自然语言处理方面的应用。

9.1 神经网络基础

仿生学是近现代科技创新的重要来源，神经网络（Neural Network，NN）是人工神经网络（Artificial Neural Network，ANN）的缩写，是以模仿大脑神经结构为出发点的计算模型。伴随着这个朴素的想法，神经网络模型几乎伴随着计算机科学一起诞生，但发展并非一帆风顺，本节先介绍 NN 的应用场景与运行原理。

9.1.1 人工神经网络

机器学习的根本目的无外乎是找到拟合已知数据的最佳方式，从而能够回答诸如数据组织关系（聚类、降维）或预测未知属性（分类、回归）的目的。在数学上，机器学习模型可以表示成一种函数关系：

$$(y_1, y_2, \ldots, y_m) = F(x_1, x_2, \ldots, x_n)$$

其中 $y_1, y_2, \ldots, y_m$ 是长度为 m 的输出向量，$x_1, x_2, \ldots, x_n$ 是长度为 n 的输入向量，不同模型的差异在于用不同的方式承载函数 F。有些模型的能力弱一些，只能表达 m 等于 1 的函数，比如基本线性回归、SVM 等，需要借助 One-VS-All 等外部机制实现多目标预测；有些模型的能力强一些，比如贝叶斯派模型中，给定任意输入可以预测任意变量的输出，但是它需要使用者完全掌控模型的内部结构。

传统神经网络是这样一种模型：输入 n 与输出 m 可以是任意长度的向量，同时使用者可以将函数 f 看成一个黑盒，不需要了解其内部逻辑。

神经网络由模仿生物大脑中“感官细胞→传导细胞→响应细胞”的三层结构发展而来，它们在神经网络中相应地成为输入层、隐藏层和输出层。通常将这些层次由输入层开始从 1 开始编号，命名为 layer1、layer2……每个神经网络的 layer1 只能是输入层，用于接收模型中的已知变量；可以有一个或若干隐藏层，它们是运算的承载者；最后一层是输出层，用于输出预测数据。

每一层由若干神经元结点构成，在预测过程中数据从高层 layer1 开始流向低层。数据流动的过程就是模型计算的过程，直至在输出层获得结果，如图 9-1 所示。

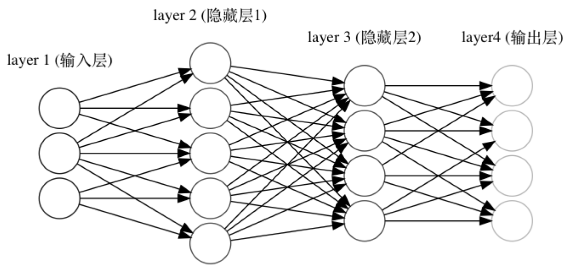


图 9-1　神经网络数据流动过程

在图 9-1 中有些神经元的数据输出到了下一层的所有神经元中，有一些则只输出到下一层的个别神经元中。减少神经元间数据的传递无疑会降低神经网络对训练数据的拟合效果，但却能降低拟合风险。特别地，如果某一层的所有结点都接收了上一层所有结点的输出，则该层网络被称为全连接层（Fully Connected Layers，FC）；如果一个神经网络的所有层都是全连接层，则该网络被称为全连接网络。

在具体应用中，输入层结点数对应于有监督学习的样本特征长度，而输出层结点对应于预测标签。比如，在一个将 28×28 像素的图像识别为 0～9 的数字 OCR 应用中，输入层需要设置 28×28=784 个结点，输出层为 10 个结点。隐藏层次数量、结点数量、连接关系的设计几乎构成了神经网络研究的核心内容，后续将逐渐展开。

9.1.2　神经元与激活函数

在介绍了神经网络的整体结构后，接下来学习网络中的每个隐藏层神经元的结构，看一看数据是如何流动的。

1. 神经元结构

如果说整个网络是一个大函数 F，那么神经元就是一个小函数 f，它的输入来自上一层结点，输出流向下一层结点。并且这个函数有固定的形式，即：

$$a_j^l = f(a_1^{l-1}, a_2^{l-1}, \ldots, a_n^{l-1}) = \sigma(w_{j1}^l a_1^{l-1} + w_{j2}^l a_2^{l-1} + \ldots + w_{jn}^l a_n^{l-1} + b_j^l)$$

其中符号 a_j^l 用于表示第 l 层第 j 个结点的激活值（activation，即结点输出值），暂且放下激活函数（activation function）σ 不讲，w_{jn}^l 是从前一层（$l-1$层）第 n 个结点连接到第 l 层第 j 个结点的传输权值（weight），b_j^l 是第 l 层第 j 个结点的偏置（bias）。所有的 w 与 b 构成了结点上的待学习参数，如图 9-2 所示。

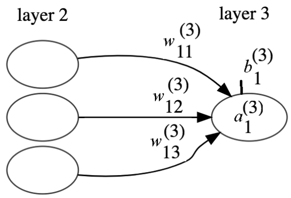


图 9-2　神经元结点（layer3 的一个结点）

图 9-2 中描述了网络中第三层第 1 个结点的权值与偏置，权值与上一层的输入一一对应，但偏置只有一个。计算后的 a_1^3 保存在结点中作为传给下一层结点的输出。

2. 激活函数

每个神经元计算的最后步骤都是通过激活函数 σ 转换权值与偏置的计算结果，其输入与输出都是一个实数。加入该激活函数有两个非常重要的目的：

◎　将神经元的输出值限定在一个区间内，使数据在网络传输中不至于变成无限大。

◎　使神经元具有非线性计算的能力，进而使整个神经网络可以处理非线性问题。

凡是能完成以上两个任务的一元函数都可以被神经网络拿来当作激活函数使用，而最经典的 σ 函数无疑是 sigmoid 函数，形式如下：

$$S(x)=\frac{1}{1+\mathrm{e}^{-x}}$$

该函数无疑是非线性的。又由于任何指数函数（包括 e^{-x}）的取值范围都是 $(0,\infty)$，因此整个 sigmoid 函数的取值范围是 $(0,1)$，如图 9-3 所示。

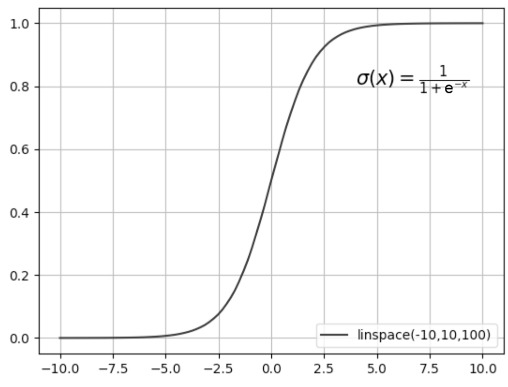


图 9-3　sigmoid 函数

还有很多其他形式的激活函数，比如 ReLU、ELU 等，后续在深度学习用到时再加以介绍。

就这样，在分层的神经网络中每个神经元完成自己的线性和非线性计算，然后传递给下一层的神经元继续计算，最后完成从输入向量到输出向量的函数变换。

注意：本节讨论的是隐藏层神经元，而输入与输出层都更为简单：输入层仅读取原始数据不做任何处理；输出层结构与隐藏层一样，但有时不使用激活函数，即仅执行线性计算。

9.1.3　反向传播

数据从输入层→若干隐藏层→输出层的计算使得神经网络天然地适合于有监督的分类/回归问题，但它的前提是网络中所有神经元的 w 与 b 均已就位，使得经过神经网络计算获得的输出向量尽量接近真实值。

1. 寻找输出层的最优 w 与 b

寻找最优 w 与 b 的目标只能通过网络训练来实现，训练的过程是一种梯度下降算法，算法依赖于以下几个基本定义。

◎　设网络层数为 L，定义输出层的损失（cost）函数为：$C=\frac{1}{2}\sum_m(y_m-a_m^L)^2$，其中 y 是训练数据的真实标签向量，a^L 是训练数据的预测标签向量。

◎　定义输出层误差（error）函数 $\delta=\frac{\partial C}{\partial \mathrm{a}}\cdot\dot{\sigma}$，其中 $\frac{\partial C}{\partial \mathrm{a}}$ 是损失函数 C 随着激活函数值 a 变化的速度，$\dot{\sigma}$ 是激活函数本身变化的速度。设激活函数的自变量为：

$$w_{j1}^l a_1^{l-1}+w_{j2}^l a_2^{l-1}+\ldots+w_{jn}^l a_n^{l-1}+b_j^l\to z_j^l$$

◎　因此 δ 的物理意义就是“损失函数随着 z_j^l 变化的速度”。

◎　由 δ 的物理意义可以获得：输出层权值 w 的梯度函数为 $\frac{\partial \mathrm{C}}{\partial w^L}=a^{L-1}\delta$，即随着 w 的变化损失函数 C 变化的程度；同理获得输出层偏置 b 的梯度为 $\frac{\partial \mathrm{C}}{\partial b^L}=\delta$。

既然已经获得了输出层损失函数 C 针对 w 和 b 的梯度函数，那么为了使 C 尽量小则只

需在训练中迭代计算梯度并让 w 和 b 在梯度相反的方向进行调整，即可寻找到最佳 w 与 b。

2. 逐层向前反馈

神经网络的输出显然不仅仅依赖于输出层的 w 与 b，之前所有隐藏层的 w 与 b 都需要通过训练进行优化。

隐藏层无法获得在该层中标签 y_m 所应该具有的真实值，因此无法获得该层的 C，也就无法像输出层一样获得误差 δ 和梯度。但是可以通过另一种方法获得 δ：

$$\delta^l = ((w^{l+1})^{\mathrm{T}}\delta^{l+1})\dot{\sigma}$$

即通过下一层的误差计算。其中 $\dot{\sigma}$ 的含义与之前一样，是当前层上激活函数的变化速度；而 $(w^{l+1})^{\mathrm{T}}\delta^{l+1}$ 是将下一层的误差变化速度反馈给了当前层。有了误差函数后，就可以向输出层一样计算当前层的权重梯度与偏置梯度了。

这样，由最后一层（输出层）开始，逐层向前反馈可以找到每一层的最优 w 和 b。因此，神经网络的这种训练方法被称为反向传播（Back Propagation）。

3. 训练算法

综合输出层和隐藏层的最优解寻找策略，可以获得如下反向传播算法。

◎ 用随机值初始化网络参数。

◎ 针对每一个训练样本：

  i. 前向计算样本预测值。

  ii. 输出层损失值、误差值，获得输出层 w 和 b 的梯度。

  iii. 反向计算所有隐藏层误差值，获得隐藏层 w 和 b 的梯度。

  iv. 将所有网络参数按梯度相反方向修改，以期降低损失值。

◎ 重复上述迭代，直到无法再使损失函数 C 的值变得更低。

可以看出这是第 3 章中的随机梯度下降（SGD）在神经网络下的应用，该方法虽然不一定能找到全局最优解，但通常情况下是足够用的。通过这套算法，在输出层计算的损失值最终驱动了网络中所有参数的调优。

9.1.4　万能网络

读者现在可能已经发现所谓的人工神经网络并不复杂：固定的层级关系、有限的神经元数量、每个神经元作简单的线性计算与激活。这种模型真的能拟合任意函数吗？

在某种程度上确实可以这样说，其背后的道理与用傅里叶变换拟合任意指数函数类似。先来看一看一元函数的情况。

1. 单个神经元能做什么

假设有一个最简单的神经网络：只有一个输入结点和一个输出结点。在该神经网络中只存在两个待求参数：输出层权值 w 和偏置 b。如果用它去拟合一个一元函数，它能做到什么呢？首先看看 w 的情况，如图 9-4 所示。

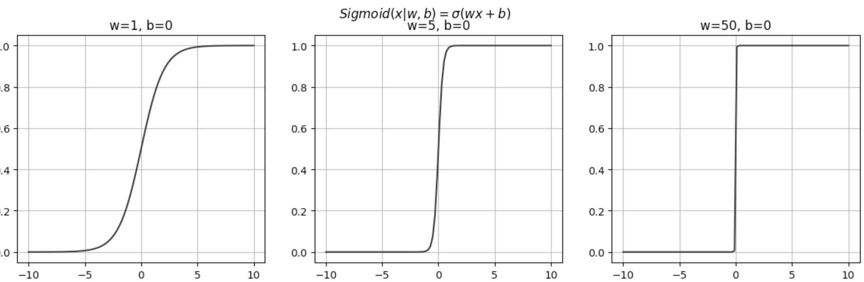


图 9-4　不同权值 w 下的 sigmoid

图 9-4 中列出了在 b 相同的情况下，w 分别取 1、5、50 所形成的变换函数。可见 w 参数调节了 sigmoid 曲线的平滑度，w 越大导致激活函数上升区域越陡峭。值得注意的是右图，该 sigmoid 函数几乎变成了一个二值函数。

再看看在 w 保持不变的情况下 b 的作用，如图 9-5 所示。

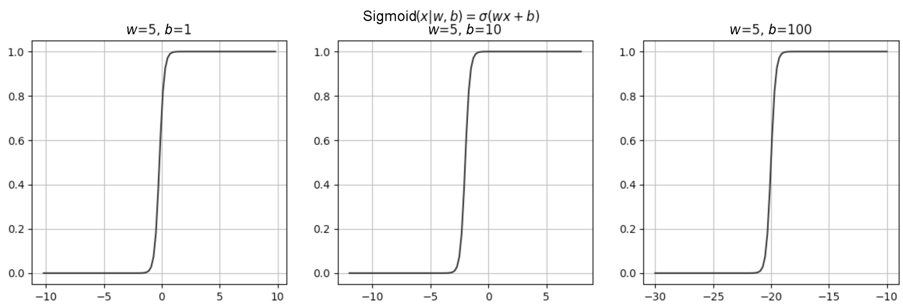


图 9-5　不同偏置 b 下的 sigmoid

在图 9-5 中，不同的 b 对函数形状影响甚少，但对该条上升曲线跳跃的位置影响很大。现在稍加想象就可以获得一个小结论：单个神经元可以近似拟合任意的单调上升（w 取正直）或下降（w 取负值）函数。

2. 两个神经元能做什么

现在假设有两个神经元处理输入变量，处理后将其传给一个输出结点，形成如图 9-6 所示的网络。

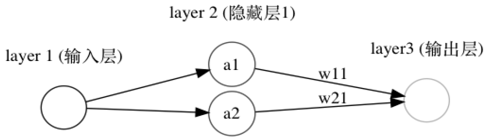


图 9-6　两个隐藏层单元的神经网络

在这样的网络中，可调节参数共有 7 个：4 条边上的权值 w、3 个神经元（隐藏层与输出层）上的偏置 b。假设忽略输出单元上的偏置 b 和激活函数，现在看看通过其他 6 个可调节参数能达到什么效果，如图 9-7 所示。

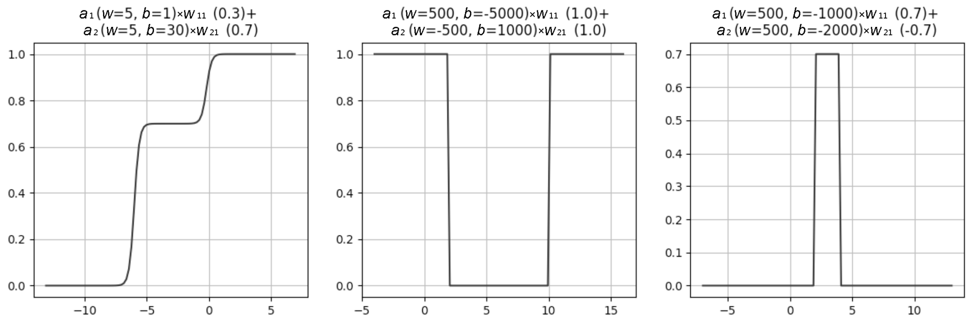


图 9-7　两个神经元的组合

在图 9-7 中列出了两个神经元结合所能产生的典型输出。虽然它们看上去不太一样，但有个共同特点是在值域内产生了两次跳变。产生不同的形状只是因为参数配置不同：左图是两个同方向 sigmoid 的结合；中图的两个隐藏单元 w 值符号相反；右图的两个输出单元 w 值相反。

通过图 9-7 的中、右两图，可以得到重要的发现：两个神经元组合可以在值域形成任意宽度、任意高度的一次脉冲。脉冲的高度由输出单元权值 w_{11}、w_{12} 定义，既可以向上、也可以向下。

3. 一个隐藏层能做什么

在神经网络的隐藏层无疑可以配置更多的神经元，那么这些神经元在一起能做什么呢？这里先给出答案：单个隐藏层可以近似拟合任意连续函数！

两个神经元配合可以形成一个脉冲的结论非常重要，这意味着任意多的“神经元对”可以两两组合在一起而不互相影响。现在假设有如图 9-8 所示的函数（左），一个神经网络（右）是如何拟合它的呢？

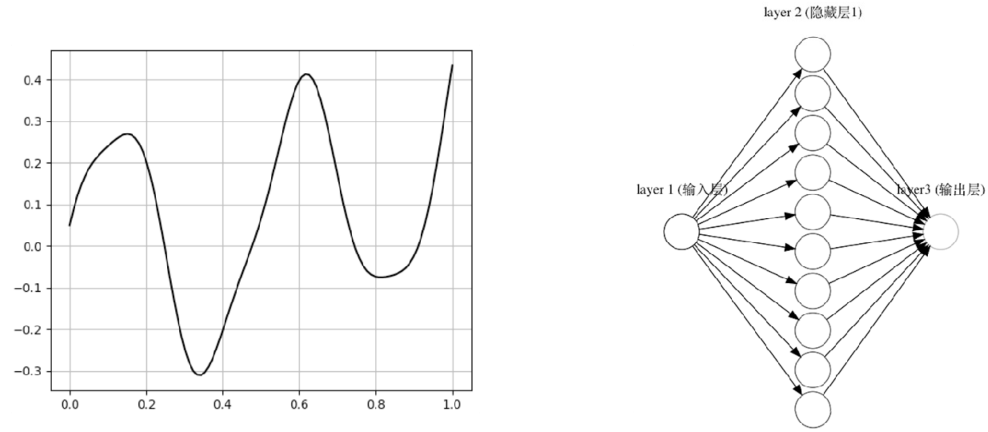


图 9-8　待拟合函数（左）与已知神经网络（右）

在图 9-8 的神经网络中共有 10 个隐藏层单元，想象一种极端情况：10 个隐藏层可以两两配合形成一个脉冲。这样就可以用 5 个脉冲拟合被寻找的函数，如图 9-9（左）所示。

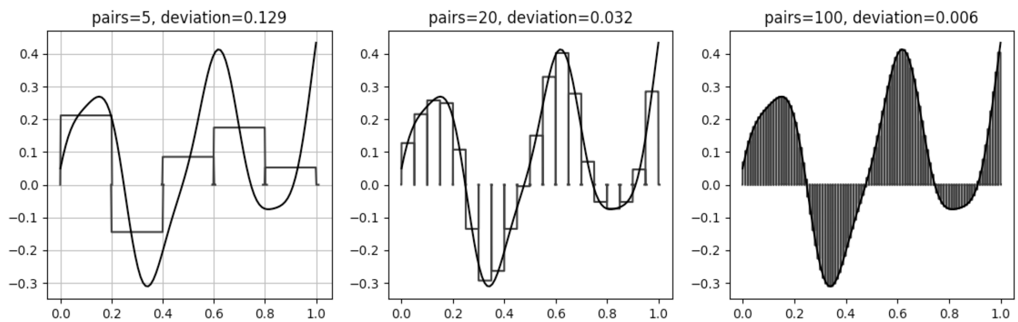


图 9-9　用脉冲拟合被寻找的函数

在图 9-9（左）中用了 5 个脉冲拟合函数，比如在[0.0, 0.2]区间内，拟合的输出约为 0.22。虽然该预测与区间内实际函数输出有较大差距，但如果逐步增加脉冲的数量，可以

发现这些脉冲一定能更好地拟合该函数，如图 9-9（中，右）所示。

图 9-9 分别列出了用 5、20、100 个脉冲拟合函数 $y = 0.3x^3 + 0.3\sin(13x) + 0.05\cos(30x)$ 的情况。图中的偏差（deviation）值是通过对值域内均匀采样 1000 个点并计算实际值与脉冲预测值之差的平均数获得的。

显然这种方法可以适用于任何一元函数，由此可以相信：一个隐藏层的神经网络可以拟合任意连续函数，拟合的精度取决于隐藏层中神经元的数量。

说明：如上实验只用于说明神经网络拟合任意函数的可能性，并非神经元在训练中真的会两两配对。一般来说总会有比这种方式更好的参数组合。

4. 多元函数会怎样

大多数的神经网络被用于解决多维特征/多元变量问题，如上针对一元函数的讨论很容易被扩展到多元函数上。

在一元函数的讨论中，拟合任意函数的关键是通过两个神经元可以组合成一个“平面脉冲”，进而由脉冲组合拟合任意函数。因此可以想象，在二元函数中如果有某种基本结构可以组合成一个“立体脉冲”，那么立体脉冲的组合仍然可以拟合任意二元函数。

有这样的基本组合吗？在 Michael Nielsen 的书籍 *Neural Networks and Deep Learning* 中对这种情况做了更细节的描述，有兴趣的读者可以参阅。这里只给出其结论：两个隐藏层的神经网络就可以拟合任意二元函数，并且该结构同样适用于更高维的情况。

至此已经可以得出结论：只要有足够的神经元，具有两个隐藏层的神经网络可以拟合任意函数，这也是为什么传统神经网络一般都是如图 9-1 所示具有 2 个隐藏层的原因。

9.2 TensorFlow 核心应用

谷歌于 2015 年开源了其诸多智能产品使用的机器学习软件库 TensorFlow。相对于 Python 的其他机器学习库，它的主要优势在于优秀的架构设计和最活跃的开发社区，因此本书选择 TensorFlow 作为深度学习实践工具，该框架可以用 pip3 工具进行管理。

```
# pip3 install tensorflow                           # 安装
# pip3 install --upgrade tensorflow==1.9.0          # 如已安装，尝试升级
```

第一条命令用于安装，第二条命令用于升级到本书使用的 2018 年 7 月发布的 1.9.0 版本。

9.2.1　张量

张量（Tensor）是在人工智能领域常被提到的一个概念，非理科出身的读者可能对这个概念比较陌生。一个常见的问题就是：什么是张量？

1. 意图

张量的概念来源于物理学中需要用与坐标无关的语言描述物理现象的需要。为什么普通的数字量（标量）不能满足这个需求呢？想象有两个观众在不同位置观看跑步比赛，观察者 *A* 在跑道的侧方，而观察者 *B* 在跑道的正前方，如图 9-10 所示。

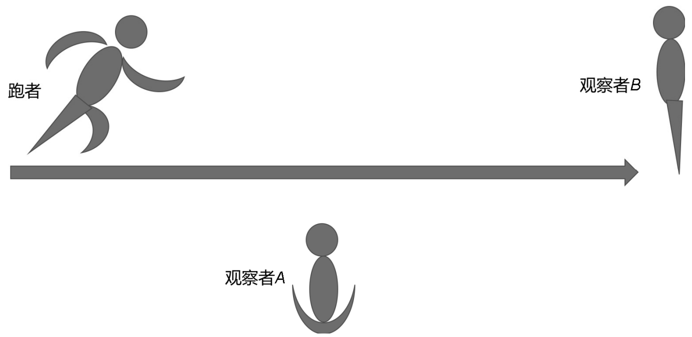


图 9-10　位置对物理现象解读的影响

设图中跑者的速度为 *s*，那么观察者 *A* 应该能更好地体会到该速度值，而在观察者 *B* 看来跑者的速度可能远小于 *s*，甚至为零。这就是为什么体育场中主席台的位置是在椭圆跑道的侧方而不是正前方的原因。

显然跑者的速度与观察者的位置无关，导致观察者 *B* 产生“错觉”的原因是，他与观察者 *A* 有不同的观察坐标系。那么应该如何描述才能使 *A* 和 *B* 都能体会到该速度值呢？物理学家想到的方式就是用更多的语言描述该速度，比如“该跑者向北的速度为 *s*1，向东的速度为 *s*2，向上的速度为 *s*3”，这样的语言不仅能适应不同位置的观察者，还能描述一个正在爬坡的跑者的移动方式，这样一组用于描述同一个物理现象的数值组合就被称为“张量”。

2. 阶

在上面的例子中，用三维空间中不同方向的值描述了速度张量的概念，其实对其更确切的说法应该是“一阶张量”。现在假设图 9-10 中的跑者在跑步过程中有一股旋风吹过，如何描述它对跑者的影响呢？如果仅用风力值描述就会犯忽略坐标系的错误，由于此时有两股力驱动着跑者（自身的跑动、旋风），最好的描述该现象的语言应该如下。

◎ 在跑者向北的运动中，旋风对其产生了推力：向北 $p11$，向东 $p12$，向上 $p13$。

◎ 在跑者向南的运动中，旋风对其产生了推力：向北 $p21$，向东 $p22$，向上 $p23$。

◎ 在跑者向上的运动中，旋风对其产生了推力：向北 $p31$，向东 $p32$，向上 $p33$。

这样用三维空间中的 3×3 个数值才完整描述了该事件。这也是一种张量，被称为“二阶张量”。这样的想法被不断扩展，张量被数学家定义成了一种多重线性映射，可以产生“零阶张量”“三阶张量”“四阶张量”……这其中零阶张量就是平时所说的标量，一阶张量就是向量，二阶张量就是矩阵。

注意：需要区分张量中“维度”与“阶数”的概念。具体地说，一个张量总由 m^n 个数值分量构成，其中 m 是维度数，n 是阶数。

3. TensorFlow 中的张量

张量是 TensorFlow 中基本的数据单元，而神经网络的运算可以看成张量在不同计算结点之间的流动，这也就不难理解 TensorFlow 名称的由来了。

虽然张量的物理与数学背景高深，但在 TensorFlow 这样的工程应用中却可以通过简单的多维数组进行表达。在 TensorFlow 中每个张量除了其包含的数值外，还有两个重要属性。

◎ dtype：数值分量的类型，比如 float32、int32、string 等。

◎ shape：张量的形状。比如[]表示零阶张量，[d]表示 d 维空间的一阶张量，[$d1$, $d2$]表示二阶张量。

TensorFlow 中允许张量各阶的维度数不同，比如在二阶张量 shape=[$d1$, $d2$]中，$d1$ 和 $d2$ 可以取不同的整数。

开发者主要通过四种方式定义 TensorFlow 中的张量：Variable、constant、placeholder、SparseTensor，代码如下。

```
>>> import tensorflow as tf                  # 引入 TensorFlow 开发包

# 零阶浮点数张量
>>> circle_rate = tf.Variable(3.14159265359, tf.float64)
# 一阶字符串张量
>>> hellos = tf.Variable(["Hello", “Hi”, “How are you”], tf.string)
# 二阶整数张量
>>> squares = tf.Variable([ [4, 9], [16, 25] ], tf.int32)

>>> print(hellos)                            # 查看张量属性
<tf.Variable 'Variable:0' shape=(3,) dtype=string_ref>
```

像 Numpy 中的 ndarray、Python 中的 list 等类似多维数组对象一样，TensorFlow 中的 tensor 也支持 slicing、reshape 等常用操作。

```
>>> squares[1, 0]                            # 读取[1, 0]索引处的分量
>>> squares[:, 0]                            # 读取第二阶索引为 0 处的分量

>>> tf.reshape(hellos, [-1, 1])              # 转换 hellos 为新形状[-1, 1]

>>> rank_three = tf.ones([3, 4, 3])          # 初始化一个全 1 的三阶张量
>>> zeros = tf.zeros(rank_three.shape)       # 初始化一个全零张量

>>> tf.enable_eager_execution()
>>> tf.executing_eagerly()                   # 开启 eagerly 模式
>>> tf.rank(rank_three)                      # 查询张量阶数
<tf.Tensor: id=11, shape=(), dtype=int32, numpy=3> # numpy=3 表示阶数为 3
```

注意：在初学或调试时开启 eagerly 模式可以随时查看 TensorFlow 表达式的执行结果，在生产环境中开发者应该编码管理 Session 的执行以提高执行效率，后续会逐步介绍。

在上述代码中，–1、冒号“:”逗号“,”“shape”等符号的意义与 Numpy 的 ndarray 相同，不再赘述。

由于 list、ndarray 等数组形式也可以用来表达张量，在 TensorFlow 中它们被称为“Tensor-like objects”，在很多接口中与 tf.Tensor 类型的参数通用。

9.2.2　开发架构

如果仅仅用“张量流”描述 TensorFlow 架构对于程序员来说太过简单，官网上如图 9-11 所示的 TensorFlow 开发栈给出了一个非常好的鸟瞰视角。

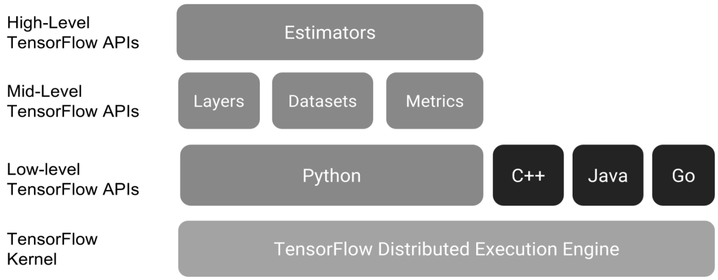


图 9-11 TensorFlow 开发栈

对于 TensorFlow 使用者来说，以图 9-11 从上到下的顺序学习是比较好的选择。Estimators 中提供了若干封装模型，在使用方式上类似于 scikit-learn；中层的 Layers、Datasets、Metrics 接口给予开发者更多、更细致的模型控制能力；底层的其他接口包括各种数学封装、硬件利用、分布式执行等接口。除了原生的 Python 接口，TensorFlow 在低层接口还支撑 C++、Java、Go 等其他语言。由于本书主题是模型与算法的应用，主要讲述的是图中左上五部分的开发应用。

9.2.3 数据管理

在神经网络的训练与验证过程常需要反复利用样本数据，TensorFlow 建立了一套基于数据集（dataset）与迭代器（iterator）的机制使得调用者可以较容易地对数据张量进行命名（Directionary Tensor）、打散（shuffle）、分批（batch）、映射（map）、复制（repeat）等管理操作。

1. 数据集与迭代器

TensorFlow 用接口 tf.data.Dataset 定义了数据集（即张量集合）类型。通过函数 from_tensor_slices 可以初始化数据集对象，比如：

```
>>> import tensorflow as tf     # 将 TensorFlow 重命名为 tf 是官方约定做法
>>> import numpy as np

>>>dataset = tf.data.Dataset.from_tensor_slices(
           np.array([[1.0, 'a'], [2.0, 'b'], [3.0, 'c'],
                     [4.0, 'd'], [5.0, 'e']]))
```

该函数将输入张量的第一个维度拆分，形成一个数据序列。在上例中，输入参数是一

个 shape==(5, 2)的二阶张量，加入数据集后则变成五个 shape==(2,)的一阶张量。如果输入的是三阶张量，数据集则是一个二阶张量序列，以此类推。

通过迭代器可以读取数据集中的张量，比如：

```
>>> sess = tf.Session()                              # 建立会话对象

>>> iterator = dataset.make_one_shot_iterator()# 建立迭代器
>>> next_element = iterator.get_next()
>>> sess.run(next_element)                           # 读取元素，并让迭代器向前
array([b'1.0', b'a'], dtype=object)                  # dataset 中的第一个张量
>>> sess.run(next_element)                           # 读取元素，并让迭代器向前
array([b'1.0', b'a'], dtype=object)                  # dataset 中的第二个张量
```

会话(Session)是管理 TensorFlow 组件执行的对象，非 eagerly 模式下必须初始化一个 Session 然后调用它的 run()函数执行 TensorFlow 运算，迭代器的 get_next()函数返回数据集中的下一个元素，将其传入会话的 run()函数执行。关于会话的作用后续有章节介绍。

注意：迭代器的 get_next()利用了 Python 的 yield 机制，对其不熟悉的读者可以想象它每次读取一个元素，如读不到则抛出异常。

2. 字典数据

如果给 from_tensor_slices 传入的是一个 Python 字典，则它要求字典中的每个 Value 必须是等长数组/张量，然后将数组中的元素按序提取逐个生成数据集中的一条数据，比如：

```
>>> dataset = tf.data.Dataset.from_tensor_slices(
 {"age":[23, 34, 30], "name":["David", "Kelly", "Rose"]})
>>> iterator = dataset.make_one_shot_iterator()
>>> next_element = iterator.get_next()
>>> sess.run(next_element)
{'age': 23, 'name': b'David'}                                  # 注意数据形式
```

用此种方式建立的数据集中的数据不再是简单张量，而是一种键值对数据。这种形式的数据为每个数值定义了名称，便于后续训练时用名称挑选数据特征、绘制图表。

还可以将字典数据以元组成员的方式传递给 from_tensor_slices()，比如：

```
>>> dataset = tf.data.Dataset.from_tensor_slices((
 {"age":[23, 34, 30], "name":["David", "Kelly", "Rose"]},
```

```
 [3000, 9000, 7000]))
>>> iterator = dataset.make_one_shot_iterator()
>>> next_element = iterator.get_next()
>>> sess.run(next_element)
({'age': 23, 'name': b'David'}, 3000)
```

此时 from_tensor_slices()分别拆分了元组中的字典和普通张量。

3. 数据变换

数据集存在的最主要意义就是它能让开发者方便地对数据进行各种变换，比如用 map()可以对所有数据进行自定义变换：

```
>>> dataset = tf.data.Dataset.from_tensor_slices(
 {"age":[23, 34, 30], "name":["David", "Kelly", "Rose"]})
>>> dataset = dataset.map(lambda x: {"age":x["age"] + 1, "name": x["name"]})
```

上述代码对数据集中所有元素的年龄 age 加 1，并保持 name 不变。还可以用 shuffle()打乱数据的原有顺序：

```
>>> dataset = dataset.shuffle(buffer_size=1000)
```

在神经网络训练中将数据分批送入可以减少内存的占用，用 batch()可以将数据划分为多个批次：

```
>>> dataset = dataset.batch(10)                # 每 10 条数据为一个 batch
```

还可以用 repeat()在数据集中“无限”复制已有数据，使迭代器读取该数据集时永远不会遇到“已读完”异常：

```
>>> dataset = dataset.repeat()
```

函数 repeat()当然不会浪费内存真的复制那么多数据，通过循环列表数据结构就足以为迭代器制造取之不尽的假象。

4. 一些扩展

TensorFlow 为数据集和迭代器都定义了一些扩展功能。

◎ 通过 Saver 对象持久化数据集到物理文件。

◎ 定义了三个数据集子类 TextLineDataset、FixedLengthRecordDataset、TFRecord

Dataset 用于从指定格式的磁盘文件读取数据。

◎ 定义了在读取时可传入变量（placeholder）的 initializable 迭代器、允许多个同构数据集同时使用的 reinitializable 迭代器、根据 placeholder 值选择不同其他迭代器的 feedable 迭代器等。

这些功能不再一一举例，读者需要时可以查阅在线文档。

9.2.4 评估器

评估器（estimator）是指 TensorFlow 中为开发者提供的一组进行了高度封装的机器学习模型，该组模型主要意义在于节省开发时间、快速验证建模可行性。

1. 使用方法

按照官方建议对它们的使用一般需要如下步骤。

◎ 编写在训练过程中产生数据集的函数，该函数可以有两种返回值形式：

i. 返回一个字典数据集（Dataset），其中的数据是二元组（features，labels）。

ii. 直接返回一个（features，lables）二元组，其中的 features 是一个特征张量数据（可以是字典形式的命名数据），labels 是标签。

◎ 用 tf.feature_column 中的函数定义一些特征对象，比如：

```
>>> age = tf.feature_column.numeric_column(‘age’)
>>> name_len = tf.feature_column.numeric_column(‘name’,
 normalizer_fn='lambda x: len(x)’)
```

上述代码定义了两个数字类型特征，其中第二个 name_len 特征是将原来的字符串类型特征转换为数字特征。

◎ 初始化一个 Estimator 对象，并指定需要训练数据集中的哪些特征，比如：

```
>>> estimator = tf.estimator.DNNClassifier(     # 神经网络分类器 Estimator
        feature_columns=[age],                  # 需要训练的特征
        hidden_units=[1000]                     # DNNClassifier 的特有参数，
定义隐藏层数量、每层的神经元数量
    )
```

如上代码初始化一个神经网络分类器（有一个隐藏层，该层有 1000 个神经元），仅训练数据集中的 age 命名特征。

◎　调用评估器方法进行训练、评估、预测。

2. 代码示例

如下代码是一个完整的用评估器进行 TensorFlow 有监督学习开发的示例：

```
    import tensorflow as tf

    # 训练数据，以二元组形式生成数据
    def input_fn():
        return {"age":[19, 23, 34, 30, 60, 55, 36],
                 "name":["Robert", "David", "Kelly", "Rose", "Hayes", "Luis",
"Lili"]},
                 [0, 0, 1, 1, 0, 0, 1]

    # 预测与评估数据， 以 Dataset 形式生成
    def input_fn_predict():
        dataset = tf.data.Dataset.from_tensor_slices((
              {"age":[23, 33, 58, 20, 59]}, [0, 1, 0, 0, 0]))
        dataset = dataset.batch(10)            # 必须分批，否则会抛出异常
        return dataset

    def main(argv):
        # 定义特征对象
        age = tf.feature_column.numeric_column('age')

        # 初始化 Estimator 对象
        estimator = tf.estimator.DNNClassifier(
            feature_columns=[age],
            hidden_units=[1000]
        )

        # 训练
        estimator.train(input_fn=input_fn, steps=10000)

        # 评估
        evaluation = estimator.evaluate(input_fn=input_fn_predict)
        print('\nAccuracy: {accuracy:0.3f}\n'.format(**evaluation))
```

```
    # 预测
    predictions = estimator.predict(input_fn=input_fn_predict)
    for idx, pred_dict in enumerate(predictions):
        class_id = pred_dict['class_ids'][0]
        probability = pred_dict['probabilities'][class_id]
        print("{}: label:{}, prob: {}".format(idx, class_id, probability))

if __name__ == '__main__':
    tf.logging.set_verbosity(tf.logging.INFO)       # 设置运行日志级别
tf.app.run(main)                                    # 运行
```

上述代码在 main()函数中依次进行了定义特征、初始化对象、训练模型、评估、预测。通常在真实环境中训练、评估、预测会使用不同的数据集，本例中因为篇幅原因只使用了两个，分别演示以元组和 Dataset 两种方式返回的 input_fn 函数。

3. 分析结果

如上代码运行后的输出为：

```
Accuracy: 1.000                             # 精确率 100%

0: label:1, prob: 0.5276459455490112        # 第 0 条数据预测标签 1，  概率 53%
1: label:0, prob: 0.5477777123451233        # 第 1 条数据预测标签 0，  概率 55%
2: label:0, prob: 0.7206478714942932        # 第 2 条数据预测标签 0，  概率 72%
…
```

其中 Accuracy（精确度）是评估器 evaluate()函数的字典类型返回值中的数据，不同的评估器（比如 xxRegressor）会有其他的评估指标。

预测函数 predict()的输出是一个迭代器，需要在循环中逐个读取预测结果。每个迭代返回的结果仍是一个字典对象，对分类器来说比较重要的键是‘class_ids’和‘probabilities’，分别表示最可能的分类和分类概率。神经网络预测中返回的概率值通常是所谓的 Softmax 值，本章后续将给予更多介绍。

说明：读者可以修改代码中 DNNClassifier 的超参数，比如减少隐藏层结点数量或反向传播训练最大迭代次数 steps 参数，看看评估精确度是否会降低。

4. 其他评估器

TensorFlow 为使用者配置了若干即插即用的评估器模型，除上文介绍的神经网络分类器 DNNClassifier 外，较重要的还有 DNNRegressor、LinearClassifier、LinearRegressor、BaselineClassifier、BaselineRegressor 等。它们的作用由命名显而易见，后两者使得开发者能较方便地定制自己的分类器和回归器。

9.2.5 图与会话

TensorFlow 用图（Computational Graph）建立计算模型，并通过会话（Session）管理图的生命周期、运行位置（比如分布在多台机器上运行）等。上一节中介绍的评估器实际上在内部管理了图与会话，但使用评估器开发者无法控制模型训练过程、无法进行并行计算。从现在开始学习开发者如何自己调用这些底层接口、进行更灵活的神经网络机器学习。

1. 从零开始

TensorFlow 把人工神经网络看成一个有向图，开发者可以用编程语言搭建任意的图结构。官网上给出的最简单的图就是一个加法运算：

```
>>> import tensorflow as tf
>>> a = tf.constant(3.0, dtype=tf.float32 , name="a")
>>> b = tf.constant(4.0, name="b")
>>> total = a + b
```

上述代码建立了一个由三个结点构成的如图 9-12 所示的图模型。



图 9-12　图模型

其中包含两个常量输入结点和一个加法计算结点。如果现在用 Python 语法显示变量 total 的值，会发现无法读到数字“7”：

```
>>> print(total)
Tensor("add:0", shape=(), dtype=float32)    # 只显示了张量
```

这是因为上面的代码只是定义了图的结构（只是定义了需要做加法），但是尚未运行（加法尚未被执行）。会话则是用来管理图的生命周期、读取执行结果，代码如下。

```
>>> sess = tf.Session()                    # 建立会话对象
>>> print(sess.run(total))                 # 执行 total 所在的图，并读取结果
7.0
```

上述代码输出了图 9-12 执行后的计算值 7.0。可以给 run()函数传入列表、字典等 Tensor-like objects，该函数会执行表达式中张量所在的图，并返回张量结果：

```
>>> sub = a - b
>>> print(sess.run({"a+b":total, "a-b":sub}))
{'a+b': 7.0, 'a-b': -1.0}
```

上述代码又定义了一个减法计算结点，并在 run()中同时读取 total 和 sub 两个结果。

2. 占位器（placeholder）

在 9.2.1 节中提到过在 TensorFlow 中定义张量的方式有 Variable、constant、placeholder 等。其中前两者比较容易理解，即变量和常量，而占位器（placeholder）又是什么呢？

其实 placeholder 也是一种变量，将其从 Variable 中独立出来与机器学习模型的开发过程有关，一般的过程为：定义模型、输入数据、训练与验证。在神经网络模型定义阶段需要定义出如图 9-1 所示的逻辑结构，而图 9-1 中的输入层结点与其他层的结点有显著区别：它们是需要在模型训练/验证/预测时待传入的数据，而隐藏和输出结点的权重、偏置、激活值等则是依赖于这些输入产生的计算中间值或结果。TensorFlow 将前者定义为 placeholder，后者定义为 Variable，而 constant 可以用来定义模型的超参数。

用 placeholder 改造加法计算、并在 run()函数中输入真实值的代码如下：

```
>>> a = tf.constant(3.0, name="a")                       # constant
>>> x = tf.placeholder(tf.float32, name="x")             # placeholder
>>> y = tf.placeholder(tf.float32, name="y")
>>> z = a + x + y                                        # Variable
>>> print(sess.run(z, feed_dict={x: 3.0, y: 4.0}))
10.0
```

与 constant 不同的是：placeholder 在对象初始化时必须传入数据类型（此处为 tf.float32），并且在执行会话 run()时用 feed_dict 参数传入真实值。在调用时也可以省略参数名 feed_dict，比如：

```
>>> print(sess.run(z, {x: 5.0, y: 4.0}))
12.0
```

3. 使用 Layers

一生二、二生三、三生万物，使用上述用 constant、placeholder、Variable 搭建的图几乎使得开发者可以随意地定义任何神经网络模型。但是神经网络毕竟有固定的模式：网络中有若干层；每一层有若干神经元；每个神经元都有若干权值、一个偏置、某个激活函数。因此 TensorFlow 在 tf.layers 中定义了若干类，使得开发者可以快速建立各种不同形式的隐藏层/输出层。

Dense 类是一种最普通的 layer，使用它可以建立一个类似图 9-1 中隐藏层 2 所示的全连接层，其常用属性如下。

◎ units：该层中神经元的数量。

◎ activation：激活函数的形式，默认为 None，即不使用激活函数。

◎ use_bias：每个神经元是否带偏置(bias)。

◎ kernel_initializer、bias_initializer：权值、偏置的初始化函数。

Dense 层的使用示例如下：

```
>>> x = tf.placeholder(tf.float32, shape=[None, 3])    # 输入层，三个神经元
>>> y = tf.layers.dense(x, units=1)                     # 输出层，一个神经元

>>> init = tf.global_variables_initializer()        # 初始化权值、偏置等变量
>>> sess.run(init)

# 输入两组数据并预测
>>> print(sess.run(y, feed_dict{x: [[1, 2, 3], [4, 5, 6]]}))
[[3.7473717]
 [9.499175 ]]
```

代码中需要注意的是：

◎ 在输入层中用 shape==[None, xx]的方式定义输入张量维度，其中的 None 代表可以一次放入任意组的输入数据进行预测。

◎ 一个 layers（此处为 dense）对象内部定义了若干权值、偏置变量，在运行计算图之前必须对这些变量进行初始化。

◎ 上述代码建立了一个只有输入层和输出层的简单神经网络，但该网络未被训练过。因此输出结果完全随机，重复运行上述代码可能会得到不同结果。

除 dense 外，TensorFlow 还在 tf.layers 中定义了很多其他形式的层次类，比如 ConvXX、AveragePoolingXX 等，这些是深度学习模型的典型隐藏层，后续章节将逐步介绍。

9.2.6　迭代（epoch）训练

通过 placeholder、layers 搭建网络图结构后，看看 TensorFlow 如何训练该网络。在 9.1.3 中已经了解了神经网络的训练算法是反向传播的，因此 TensorFlow 围绕损失与优化的概念进行模型训练：

◎　损失（loss）是衡量当前模型预测值与真实值之间差距的函数，比如可以用所有样本预测值与真实值的平方差均值衡量。

◎　优化（optimize）则是寻找最低 loss 的算法。

TensorFlow 在 tf.losses 和 tf.train 中预定义了很多损失函数与优化器，如下代码使用其中的 GradientDescentOptimizer（梯度下降）优化 mean_squared_error（平方差均值）损失，用一个神经网络拟合函数：

$$y = \left(x_1 + \frac{x_2}{2}\right) + 1$$

```
import tensorflow as tf

##################### 定义图结构 ######################
# 输入层 2 个神经元
x = tf.placeholder(tf.float32, shape=[None, 2])
# 3 个神经元的隐藏层，使 sigmoid 激活函数
hidden =  tf.layers.dense(x, units=3, activation=tf.nn.sigmoid)
# 输出层 1 个神经元
y = tf.layers.dense(hidden, units=1)

##################### 定义损失函数、优化器 ######################
# 真实输出的 placeholder
y_true = tf.placeholder(tf.float32, shape=[None, 1])
# 损失函数
loss = tf.losses.mean_squared_error(labels=y_true, predictions=y)
optimizer = tf.train.GradientDescentOptimizer(0.01)    # 梯度下降优化器
train = optimizer.minimize(loss)

##################### 初始化所有变量 #####################
```

```
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

##################### 训练 ##############################
for i in range(2000):
  _, loss_value  sess.run((train, loss),
    {x: [[1, 1], [2, 2], [1, 2], [2, 1]],              # 运行时传入训练数据
     y_true:[[2.5,], [4,], [3, ], [3.5, ]]})
  print(loss_value)

##################### 预测、验证 ##########################
print(sess.run(y, feed_dict={x: [[1.5, 2]]}))
```

代码分析如下。

◎ 代码中未输入任何公式 $y=\left(x_1+\frac{x_2}{2}\right)+1$ 的计算逻辑，只是在训练阶段用 feed_dict 形式输入了样本数据。

◎ 训练数据的特征与真实标签值都是输入数据，代码为它们定义了 placeholder 对象 x 和 y_true。

◎ 在定义损失函数时给出比较的变量：真实标签值(y_true)与预测标签值(y)。

◎ 初始化 GradientDescentOptimizer 时定义梯度下降的学习速度(learning rate)为 0.01。该值的意义可参考 9.4.3，总体来说该值越大学习速度越快。

◎ 用一个循环不断地执行优化器进行反向传播训练，并读取损失值。

在神经网络开发中，将把所有数据都训练了一次的行为称为一个 epoch（比如本例中 for 循环内的 run()调用）。这种迭代调用优化器进行训练的方式使得开发者有机会控制、监测整个训练过程：

◎ 可以直接根据当前损失值 loss_value 判断是否需要继续训练。

◎ 可以用验证数据检验模型的精确度，然后判断是否需要继续训练。

◎ 可以抓取当前模型状态，进行后续分析等。

本例中仅采用了最简单状态抓取：对每个 epoch 进行训练后，用 print 函数输出当前训练样本的损失值。输出结果如下：

```
7.2761807                                              # 第一个 epoch 的 loss
6.690956                                               # 第二个 epoch 的 loss
6.145658                                               # ……
…
0.05620346
0.056157842
```

损失值随着优化器的不断执行逐渐降低，在 2000 次迭代后达到 0.056157842。

再看一看本例代码中最后用模型预测 y 变量的结果是：

```
[[3.3707757]]
```

用代码中的输入数据 $x1 = 1.5$，$x2 = 2$ 带入公式 $y = \left(x_1 + \frac{x_2}{2}\right) + 1$ 可以得到正确值是 3.5。可见本例产生的模型预测(3.3707757)已经与真实值非常接近，读者可以尝试增加 epoch 数量，看是否会得到更好的预测结果。

9.2.7　图与统计可视化

到目前为止接触到的计算图都是抽象的代码，当计算流程变得复杂时这些代码不便于思考和呈现。既然它被命名为“图”，TensorFlow 一定开发了配套工具将其可视化。

1. 生成可视化文件

TensorFlow 用 tf.summary.FileWriter 对象将“代码图”转换为“可视图”。其实只要在基于图与会话开发的程序中加入两行代码就可以将计算逻辑绘制到网页中，比如：

```
# 建立 FileWriter 对象
>>> writer = tf.summary.FileWriter("./log/", sess.graph)
>>> writer.close()                                   # 也可以调用 writer.flush()
```

在初始化会话对象（本例中为 sess）后即可新建 tf.summary.FileWriter 对象，其中传入的两个参数是：

◎　可视文件保存的文件系统目录。

◎　图对象，传入 sess.graph 即可。

如果在 FileWriter 初始化时未传入图对象，也可以之后通过 writer.add_graph()方法添

加。在调用 writer.close()或 writer.flush()后，对象会在指定目录（本例为（"./log/",）中生成形如 events.out.tfevents.xxx.yyy 的可视化文件，其中 xxx 是生成时间戳，yyy 是当前主机名。

2. 显示可视化文件

用 FileWriter 生成的 events.out.tfevents.xxx.yyy 文件并非 PNG、GIF 等图形文件，需要使用随 TensorFlow 安装的 TensorBoard 工具读取，即在控制台中执行如下命令：

```
# python3 -m tensorboard.main --logdir ./log

TensorBoard 1.9.0 at http://hostname:6006 (Press CTRL+C to quit)
```

其中“—logdir”参数需要给出可视化文件所在的目录。该命令启动了一个 HTTP 服务器，等待使用者从 6006 端口访问浏览可视图（如其在输出中提示）。因此该命令并不会退出，直到收到控制台 CTRL+C 命令。现在打开 IE/Chrome 等任意浏览器导航到 http://localhost:6006，就可以查看计算图了。

如果在代码中未给定义的张量指定名称，TensorFlow 会根据自己的规则为每个张量和运算定义名称，使得最终显示的图易于理解，因此为所有张量在初始化时配置 name 属性是一个好的习惯。

3. 实例：图结构可视化

改造上一小节的程序，使其能较好地可视化图结构，完整代码如下：

```
import tensorflow as tf

x = tf.placeholder(tf.float32, shape=[None, 2], name="input_layer")
hidden =  tf.layers.dense(x, units=3, activation=tf.nn.sigmoid,
              name="hidden_layer")
y = tf.layers.dense(hidden, units=1, name="output_layer")
y_true = tf.placeholder(tf.float32, shape=[None, 1], name="y_true")

loss = tf.losses.mean_squared_error(labels=y_true, predictions=y)
optimizer = tf.train.GradientDescentOptimizer(0.01, name = "optimizer")
train = optimizer.minimize(loss, name="min_loss")

init = tf.global_variables_initializer()
```

```
with tf.Session() as sess:
  writer = tf.summary.FileWriter("./log/", sess.graph)
writer.close()
```

本例中为所有张量定义了 name 属性，并使用 FileWriter 生成了图结构。在启动 TensorBoard 工具后，打开浏览器可以得到如图 9-13 所示的可视化图。

注意：因为图的静态结构不取决于该图是否已经运行，所以上述代码无须执行 sess.run() 函数就能生成可视化文件。

左侧的“Main Graph”是主数据流图，右侧的“Auxiliary Nodes”是对一些子计算结点的细节描述。

图中的每个结点（node）都是一个运算（operation），结点之间的边（edge）是运算之间传递的张量（Tensor）数据。TensorBoard 图数据通常从底部沿着箭头向上传递，在图 9-13 中可以清晰地看到数据 input_layer→hidden_layer→output_layer 的传递过程，最终优化器综合所有变量的结果训练网络参数。

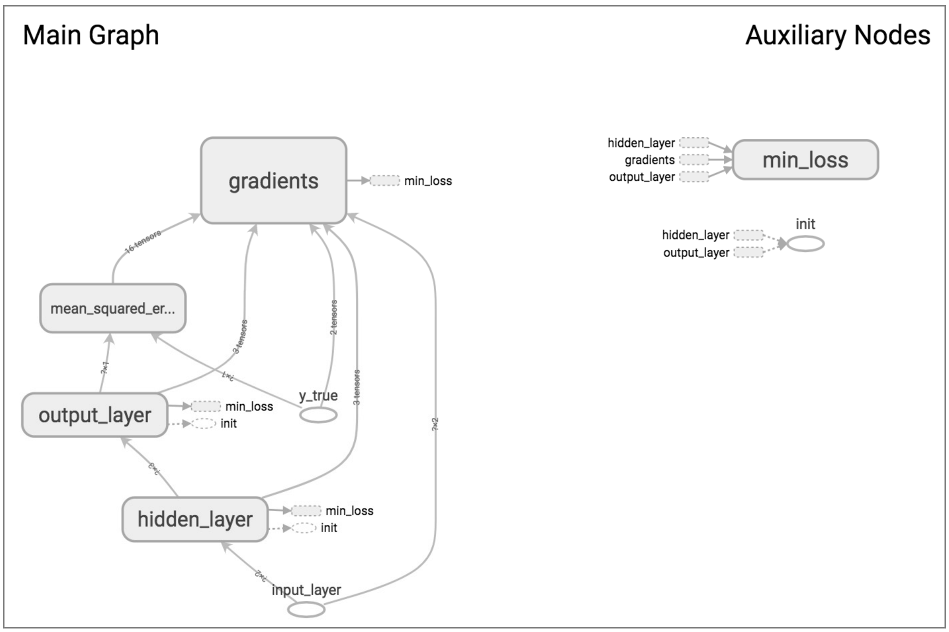


图 9-13　TensorBoard 可视化图

同时，TensorBoard 还是一个可交互的工具，比如用鼠标单击“min_loss”结点后可以

显示该结点的详细信息，如图 9-14 所示。

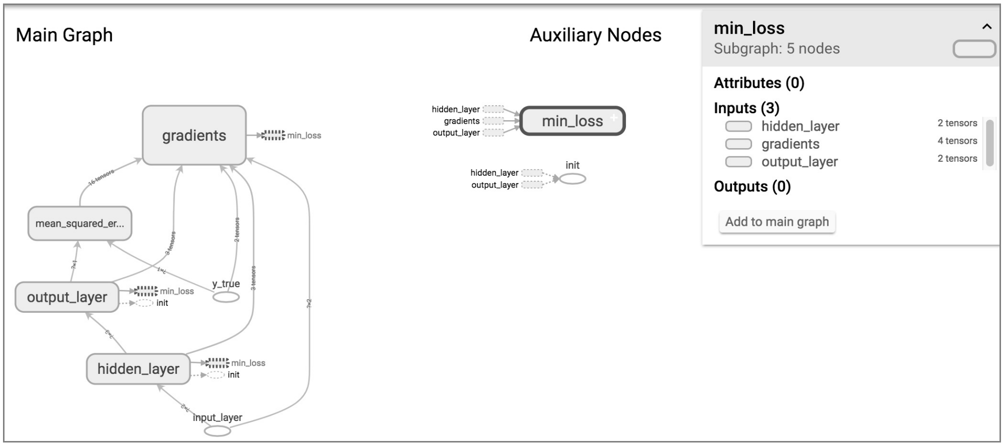


图 9-14　TensorBoard 交互举例

在图 9-14 右侧的信息卡中显示了结点的所有输入与输出张量信息，并可以用“Add to main graph”按钮将该结点加入左侧主流程图中。丰富的交互性正是 TensorBoard 采用网页而非静态 PNG 图像进行可视化的原因。

4. 统计数据可视化

除了可视化静态图结构，TensorBoard 的另一个重要功能是可视化机器学习过程中的各种统计数据。TensorBoard 在 tf.summary 包中定义了若干函数与对象支持统计可视化，常用数据类型主要有两种：Scaler 和 Histogram，前者统计单个数值随着 epoch 变化的情况，后者统计某组数据随着 epoch 变化的直方图分布情况。

网络的损失值（loss）随着 epoch 逐渐变化，是典型的 Scaler 类型数据。可用如下代码将其加入 TensorBoard 中：

```
>>> # 此处省略若干图定义代码
>>> summary_loss = tf.summary.scalar('loss', loss) # 建立一个 Scalar 统计项
>>> for i in range(2000):
        # 在图运行时传入统计项变量
        s_loss= sess.run((summary_loss,), feed_dict=xxx)
        # 统计结果加入 FileWriter
        writer.add_summary(s_loss, global_step=i)
```

上述代码首先用已定义的某个张量（本例为 loss）初始化一个 Scaler 统计项（本例为

summary_loss），然后在图运行的每个 epoch 中传入该统计项，并把结果（本例为 s_loss）用 add_summary()函数加入 FileWriter 对象中。本例最终生成的 TensorBoard 统计如图 9-15 所示。

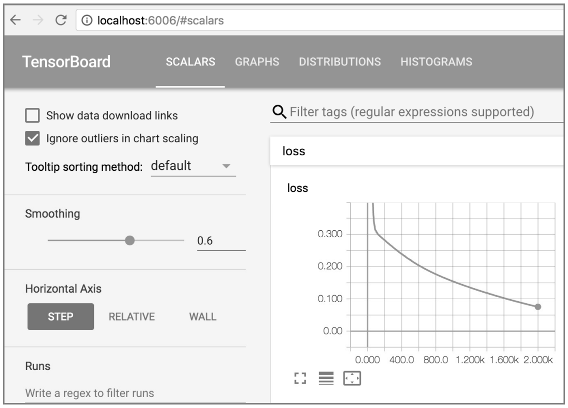


图 9-15　Scaler 统计示例

在图 9-15 中 TensorBoard 用曲线图的方式记录了 loss 值在 2000 个 epoch 中的变化过程。

5. 直方图可视化

直方图（Histogram）是除 Scaler 外另一种重要的统计图，用于可视化一组数据中各取值结果出现的次数。

神经网络每个隐藏层都有多个神经元 bias，如下代码演示对某一层的 bias 实现直方图可视化：

```
# 定义某隐藏层
>>>layer1 = tf.layers.Dense(units=3, activation=tf.nn.sigmoid,
                name="hidden_layer")
>>> hidden = layer1(x)

# 生成 histogram 统计项，参数重传入的 layer1.variables[1]是 bias 张量
>>> summary_bias = tf.summary.histogram('hidden_bias',
                        layer1.variables[1])

# 在 epoch 中将统计结果加入 FileWriter
>>> for i in range(2000):
        s_bias = sess.run((summary_bias,), feed_dict=xxx)
        writer.add_summary(s_bias, global_step=i)
```

本例中值得注意的是。

◎　本例前两行代码等价于之前案例中的如下语句：

```
hidden =  tf.layers.dense(x, units=3, activation=tf.nn.sigmoid,
 name="hidden_layer")
```

实际上 dense 类是对 Dense 类的一种快捷封装（注意两个类名的大小写区别），此处使用 Dense 类的版本是为了能从层对象（本例为 layer1）中引用 bias 张量。

◎　一个层对象内实际上封装了权值、偏置变量，可以通过 Dense 对象的 variables 属性读取它们。该属性是一个类似下面内容的二元组：

```
[<tf.Variable 'hidden_layer_1/kernel:0' shape=(2, 3) dtype=float32_ref>,
 <tf.Variable 'hidden_layer_1/bias:0' shape=(3,) dtype=float32_ref>]
```

其中第一个元素是权值张量，形如(m, n)，m 是输入神经元数量，n 是当前层神经元数量。第二个元素就是偏置张量，形如$(n,)$，即每个分量分别属于当前层的每一个神经元。

在将偏置张量配置到统计变量 summary_bias 后，就可以向之前的 Scaler 一样在每个 epoch 里将数据结果加入 FileWriter 中了，效果如图 9-16 所示。该图是一个三维制图：

◎　横坐标是统计张量中各分量的取值范围。

◎　纵坐标是每个取值所对应的分量数量，即直方图的高度。

◎　深度坐标是 epoch 编号。

如此分析图 9-16，得到的信息是：在最早的 epoch 中 bias 分布有两个集中点，分别在 –1 和 0.05 附近；随着迭代的进行，bias 逐渐集中到 0 和 0.2 附近。

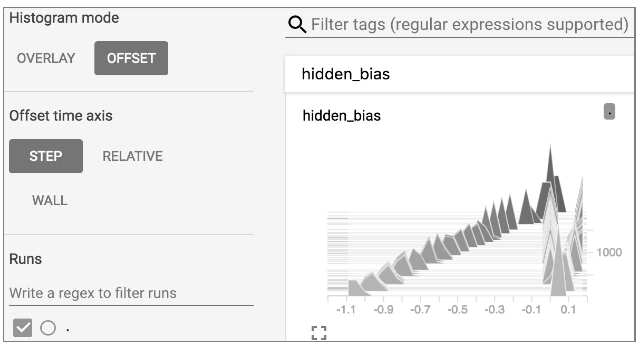


图 9-16　Histogram 统计图示例

至此已经完成了进入深度学习必备的 TensorFlow 工具方面的基础学习。本节未涉及的其他 TensorFlow 较重要的知识点包括。

◎　更好地利用硬件资源：用 GPU、TPU 等取代 CPU 计算。

◎　分布式计算：搭建 TensorFlow 集群及在其上分配任务。

脱胎于谷歌的 TensorFlow 在这些方面显然处于行业内领先地位，但由于本书的主题是模型与算法，对它们不再做详细介绍。

9.3　卷积神经网络

虽然理论上可以拟合任意函数，神经网络很长时间里在应用上的实际效果一直无法超越决策树、SVM、甚至线性回归等简单模型。在一段很长时间的沉寂之后，2012 年 Alex 发布的 AlexNet 卷积神经网络（Convolutional Neural Network，CNN）像是神经网络应用黑夜中的一颗启明星，其在图像识别 Top5 级别的错误率比当时的非神经网络方法降低了近一半，就此掀起了以神经网络为基础的深度学习应用热潮。

9.3.1　给深度学习一个理由

所谓深度学习网络（Deep Neural Network，DNN），可以简单理解为层次数量较多的神经网络。理论上两个隐藏层的神经网络就可以拟合任意函数，那么为什么需要研究层次数更多的网络呢？

1. 为什么深度学习

这个问题没有明确的答案，笔者认为主要有如下两个原因。

◎　“浅度学习”对每一层的神经元数量要求过高：每层网络对函数的拟合效果正比于神经元的数量，但对于复杂函数在每一层需要非常多的神经元才能达到较好效果。因此“深度学习”实际上是一种“虽然层次加深，但每层神经元数量较少，使得整个网络需要训练的参数总数减少”的网络搭建方法。

◎　“浅度学习”无法对学习到的知识进行分层：以图像识别为例，输入层的数据是逐像素的点阵信息，则第一隐藏层学到的知识只能是像素级别的，第二隐藏层知

识则略微高级一些，无法满足更深层次信息的学习。

如上两点原因的后者尤为重要，在某种程度上是后者导致了前者：正是因为浅层网络无法分层表达知识，所以才需要在每一层有更多的神经元。

2. 知识是有层次的

打个比方，思考一个“识别图像中是否有汽车”的任务。深度网络若要完成这个任务，可以按知识层次依次识别：

◎ 读取所有像素，判断是否有各种矩形、圆形、玻璃、金属、橡胶。

◎ 是否有车窗（方形的玻璃）、门把手（小长方形金属）、车轮（圆形橡胶）。

◎ 是否有多个车轮、有车门（大矩形金属板上有门把手）。

◎ 满足如上所有条件，则认为找到汽车。

如此，每一层只是在上一层的基础上完成一点相对简单的任务，最终得到较好的结果。

若将同样的任务交给浅层网络，对它的要求则是从杂乱的像素数据中一下子识别汽车，这常导致网络对数据的过度拟合（Over Fitting）。比如若所有汽车样本的第 *n* 个像素都是黑色（碰巧样本中车轮都在该位置），网络可能会认为所有该位置是黑色的图像都是汽车（即使对于全黑图片来说）。这样的任务对训练数据的多样性、平衡性有太高的要求了！

3. 深度学习难点：梯度消失与梯度爆炸

已经了解了建立深层次网络的必要性，那是否只要在搭建网络时简单加入更多层次就实现了深度学习呢？

当然也不是这样的，否则就小看了该领域的科学家这么多年的努力。简单叠加的网络是非常难于训练的，其主要问题在于所谓的梯度消失（Vanishing Gradient Problem）和梯度爆炸（Exploding Gradient Problem）。

神经网络的训练算法是逐层反向传播的随机梯度下降，如图 9-17 所示是根据本章 9.1.2 节中公式与流程演示的一个链状网络的反向传播示意图。

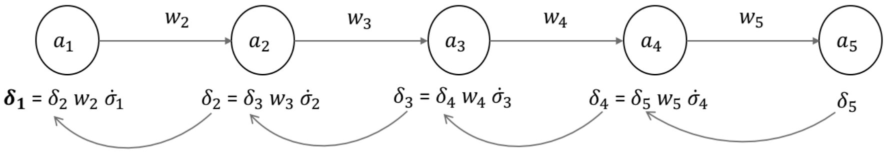


图 9-17　链状网络的反向传播示意图

在图中梯度值 δ 在网络中从后向前传递，每次传递都被权值 w 和激活函数变化率 $\dot{\sigma}$ 倍增。不难想象，较前端的神经元 a_1 学到的梯度值 δ_1 在经过一系列的 w 与 $\dot{\sigma}$ 传递而来后与原始梯度 δ_5 会有怎样大的差别。

当网络中每一层的结点数不止一个的时候，梯度下降的过程将受到更多网络参数的影响。这种千山万水传递调整梯度的方法导致的一个直接结果，就是每次梯度调整时网络前端的梯度已经几乎与原始梯度无关，反而取决于网络中现有的这许多 w 与 $\dot{\sigma}$ 。当 w 与 $\dot{\sigma}$ 较小时，梯度值 δ 随着反向传递逐渐变小，最后变化几乎停止，该问题被称为梯度消失；相反，当 w 与 $\dot{\sigma}$ 较大时梯度值 δ 呈指数级不断增大，被称为梯度爆炸。

梯度消失与爆炸的存在使得研究者们需要在网络结构、激活函数形式等方面进行探索，寻求更有效的深度学习方法。

9.3.2　CNN 结构发展

卷积神经网络（CNN）是一种通过设计特殊的（即非全连接）层间关联结构，达到降低学习参数（权值、偏置）总数，使得深度网络变得可训练的深度学习模型。如图 9-18 所示是一个最简单但足够典型的卷积神经网络结构。

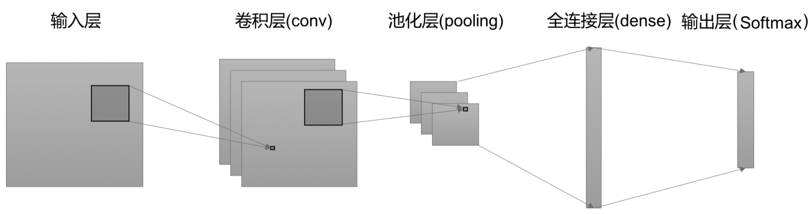


图 9-18　典型的卷积神经网络

该神经网络用于完成图像识别工作，输入层是一张图像（每个像素灰度值是一个神经元），经过若干隐藏层后到达输出层。该图中输入层、全连接层、输出层等都是读者熟知的概念，而 CNN 特有的概念是卷积层和池化层。

◎　卷积层：每个神经元仅由输入层某个局部区域计算而来。

◎　池化层：对卷积数据做固定规则压缩。

从意图上讲，卷积层可以将低层次的信息（比如像素）转换成高层次的信息（比如局部区域）；而池化层仅仅是用于减少神经元的总数，便于后续处理。此外，在图 9-18 的输出层出现了修饰词“Softmax”，它实际是一种激活函数，用于很多网络的输出层。

在一个卷积神经网络中，卷积层、池化层、全连接层等可以多次出现，以达到分层提取信息、深度学习的目的。在进一步剖析这些特殊层次与激活函数之前，先预览近年来几个里程碑式的卷积神经网络架构。

1. LeNet-5（1998）

这是学术界公认的卷积神经网络先驱，由 LeCun 等人于 1998 年在论文 *Gradient-based learning applied to document recognition* 中发布，其结构如图 9-19 所示。

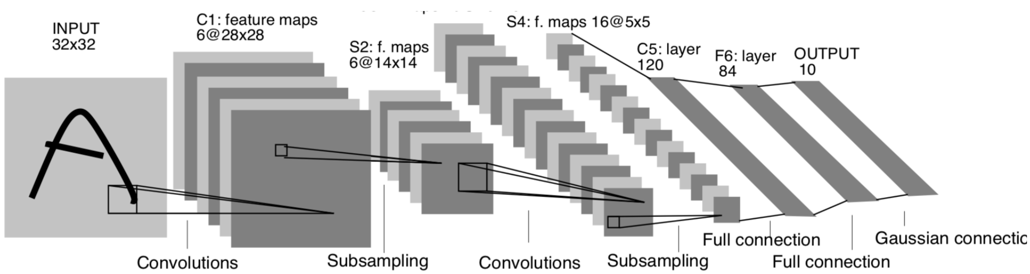


图 9-19　LeNet-5 网络结构（取自 LeCun 论文）

LeNet-5 是一个 8 层网络，被用于识别手写文字。其结构为：32×32 像素图片输入→一组卷积-池化层→另一组卷积-池化层→两组全连接层→输出层。由于当时硬件计算能力有限，该网络无法处理更高分辨率的输入图片或者让网络变得更深，所以当时并未获得过多关注。

2. AlexNet（2012）

自 2010 年起，开源图像数据库组织 ImageNet 每年举办一次图像识别竞赛 ILSVRC (ImageNet Large-Scale Visual Recognition Challenge)。ILSVRC 的目标是正确识别、定位 ImageNet 图像库中的物体或场景，至此图像识别领域有了一个统一的评比标准，该竞赛的胜出模型往往能引领后续的研究方向。

2012 年的胜出模型是深度学习卷积神经网络 AlexNet，其结构如图 9-20 所示。

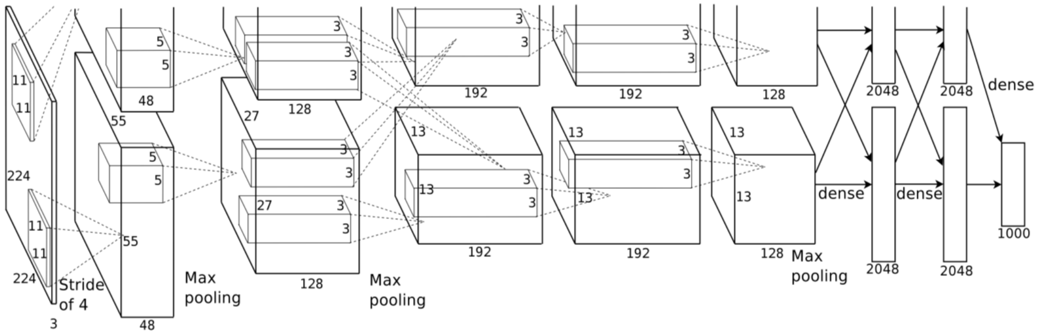


图 9-20　AlexNet 网络结构（取自 Alex 论文）

相对于 LeNet-5，AlexNet 是一个支持更高分辨率的图像输入、更深更复杂的网络。如图 9-20 所示，它将输入图像分为两部分，使得两个 GPU 可以并行运算。除了结构上的规模化，AlexNet 的重要意义还在于：

◎ 用激活函数 ReLU 解决了 Sigmoid 等激活函数的梯度消失/爆炸问题，并且训练速度更快。

◎ 用人工训练数据扩展（data augmentation）的方法扩充数据集，使得训练出的模型泛化能力更强。

◎ 在运行时随机对 0.5%的神经元进行剪枝（dropout），使得网络鲁棒性更强。

所有的这些改进最终使得 AlexNet 在 ImageNet 上 top5 级别的错误率降低到 15.4%，而当时位于第二名的模型该项错误率仅为 26.2%。

3. GoogleNet/Inception 与 VGGNet（2014）

在 AlexNet 之后深度学习进入了高速发展阶段，2014 年 ILSVRC 竞赛的前两名 GoogleNet（又名 Inception）和 VGG 是 CNN 的另一次阶段性成果，其在论文中给出的网络结构如图 9-21 所示。

如图 9-21 所示，近 100 层（有一些是并行层）的网络结构已使得横向 A4 纸面无法清晰显示。但通过色块的表示可以发现，GoogleNet 的主要结构是不断地堆叠卷积层与池化层，该网络在 ImageNet 上的 top5 错误率已经降低到了 6.67%，与普通人类识别的错误率相当。

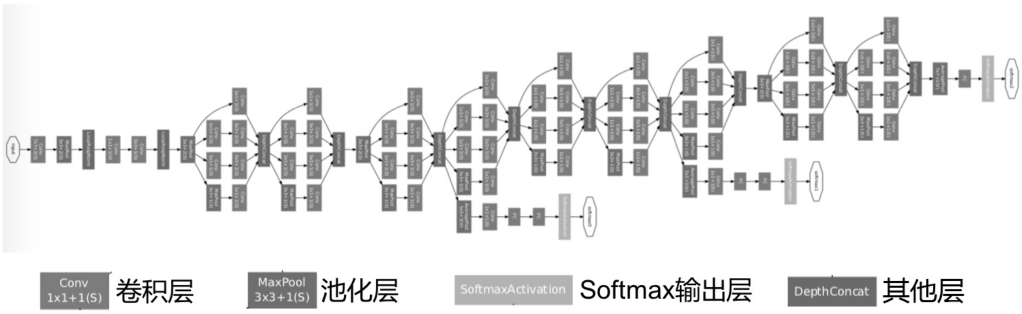


图 9-21　GoogleNet（取自 Szegedy 等的论文）

注意：2014 年的 GoogleNet 上有三个输出层，作用是为了在训练时可以利用它们同时做反向传播，加快优化速度。该方法后来随着 ResNet 的提出不再被使用。

2014 年 ILSVRC 的亚军是 19 层的 VGGNet，其特点在于用几乎处处相同的最简单卷积-池化层完成整个网络搭建。或许是大道至简，虽然 VGGNet 在 ImageNet 上表现不如 GoogleNet，但后来在社区的实际应用上却获得了更高声誉，在生活中的普通图像上体现了更高的泛化能力。

4. Google 与 Microsoft 的竞争（2015~现在）

在 2014 年以后，卷积神经网络的研究似乎已经成为了一种积木搭建游戏，不断有新的形式出现并在测试集上获得更好的效果。其中有影响力的是微软的残差网络 ResNet 和谷歌的 Inception（包括后续的 Xception）系列。前者在 2015 年的 ILSVRC 上以 3.6%的错误率获得冠军，后者于 2016 年发布了 Inception-v4，达到了 3.08%的错误率。随后谷歌在 Inception 中加入了微软 ResNet 元素，发布了 Inception-ResNet 系列，但除能加快训练速度外从实际效果上看并未有大的突破。

9.3.3　卷积层

已经了解了 CNN 的宏观网络结构，现在开始拿起放大镜看看在所有 CNN 中起关键作用的卷积层（convolution）、池化层、Relu、Softmax 等究竟是什么。

1. 卷积结构

卷积层与不同全连接层的区别可以归结为以下三点：

◎　卷积层内由多组 feature map 组成，每个 feature map 有相同的神经元数量。

◎　一个 feature map 上的每个神经元只接收上一层某个局部区域神经元的输入。

◎　一个 feature map 上的所有神经元共享同一套权值、偏置。

上述第一、二点使得卷积层具备知识分层学习的能力；第二、三点使卷积层需要寻找的参数得以大大降低。这样说来有些抽象，请观察如图 9-22 所示的卷积层结构。

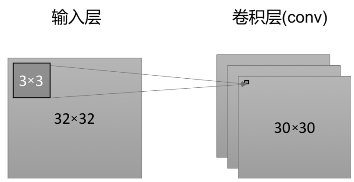


图 9-22　卷积层结构

如图 9-22 所示的输入层是一个 32×32 像素的图像，卷积层由 3 个 feature map 组成，输入层上局部的 3×3 个像素对应 feature map 上的一个神经元。

◎　输入层上像素[0, 0]~[2, 2]共 9 个值是卷积层上第[0, 0]个神经元的输入。

◎　计算窗口从[0, 0]向右移一格：输入层上像素[0, 1]~[2, 3]共 9 个值是卷积层上第[0, 1]个神经元的输入。

◎　计算窗口从[0, 0]向下移一格：输入层上像素[1, 0]~[3, 2]共 9 个值是卷积层上第[1, 0]个神经元的输入。

◎　以此类推，输入层 32×32 个像素共可以生成 30×30 个卷积层神经元。

算上 3 个 feature map，这样在本例的卷积层共有 3×30×30 个神经元。虽然神经元的数量很多，但由于同一 feature map 的神经元共享同一套权值、偏置，其实整个卷积层的待优化参数远小于简单的全连接层。

以图 9-22 为例，卷积层的每个神经元有 9 个输入权值和 1 个偏置，所以一个该图中卷积层总的待调节参数是 $(9+1)\times 3=30$ 个。而如果使用一个具有 100 个结点（该数量已经远小于输入神经元）的全连接层，待调节参数将达 $(32\times 32+1)\times 100=102500$ 个，多了 3400 倍！

2. 卷积超参数

了解卷积结构后，就不难理解大多数深度学习开发框架需要配置的卷积层超参数了。

◎ 权值张量形状（shap for weights）：权值是一个四阶张量，分别配置权值计算窗口的长、宽、上一层的 feature map 数量、本层的 feature map 数量。

◎ 偏置张量形状（shape for bias）：偏置是一个一阶张量，即需要配置本层 feature map 数量，该值必须与权值张量形状的最后一个数值相等。

◎ 步长（stride）：在图 9-22 中完成一个计算窗口后，后续窗口需要向右/向下移多少个像素。

读者可以尝试在图 9-19、图 9-20 中寻找上述超参数。

3. 卷积释意

卷积层被规范成如此结构并非毫无道理。卷积层每个 feature map 的第[j, k]个神经元激活值计算公式为：

$$\sigma\left(\sum_{l}\sum_{m} w_{l,\,m} a_{l+j,\,m+k} + b\right)$$

其中 w, b 分别是权值和偏置，a 是本层输入。对比本公式与第 2 章中给出的数学上卷积的定义，可以发现其就是权值与输入信号的卷积运算。这也是卷积神经网络名称的由来。

即使不理解数学上卷积的定义，从直观的角度可以将卷积层的每一个 feature map 想象成拿着一个放大镜（计算窗口）在地面上（输入层）搜索某个物件（高层知识、模式），而多个 feature map 其实就是多个放大镜在搜索不同的知识。

9.3.4 池化层

在之前介绍的 CNN 整体结构中，在卷积层后都跟随着一个池化层（pooling），它的作用在于进一步简化卷积层计算的数据、减少神经元数量。

从图 9-18 上看，池化层与卷积层类似：本层的一个神经元都只接收上一层局部区域神经元的数据，其不同点在于：

◎ 上一层神经元与池化层神经元是 N:1 关系，即任何一个上一层神经元仅“映射”到池化层的某一个神经元上，不似卷积层那样进行地毯式搜索。

◎ “映射”的规则非常简单：无须进行权值、偏置、激活等运算，仅采用诸如取局

部区域最大值（即 max-pooling）、平均值（avg-pooling）等简单策略。

如图 9-23 所示是一个 max-pooling 的示意图。

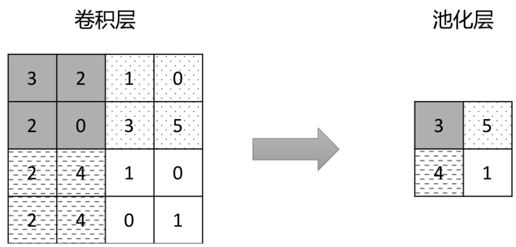


图 9-23　max-pooling 示意图

图中的输入层（通常是卷积层）是一个 4×4 的 feature map，其中每个 2×2 的局部区域被输入池化层中，输入的值是该区域中最大的一个激活值。

在前向传播（即数据从输入层流到输出层的过程）中池化层就是如此简单的计算逻辑，唯一需要思考的是在训练的反向传播过程中并非通过权值进行关联的池化层如何将损失值传递给卷积层。对于 max-pooling 来说，通常可以只向卷积层中产生最大值的神经元传递损失值，而对于 avg-pooling 来说需要对所有卷积层神经元传递损失值。

9.3.5　ReLU 与 Softmax

在 AlexNet 网络中，现在只有 ReLU 和 Softmax 两个激活函数尚未介绍，它们被分别用在 CNN 的卷积层和输出层上。

1.　Sigmoid 与梯度消失

在 9.3.1 节中介绍过传统神经网络无法更深的重要原因：损失反向传播过程中的梯度消失。这是因为在反向传播中梯度不断被乘以因子 w 和 $\dot{\sigma}$，其中 $\dot{\sigma}$ 是激活函数的导数。如果 $\dot{\sigma}$ 小于 1，则造成梯度越来越小以致消失。

而传统激活函数 Sigmoid 的导数是什么样子呢？如图 9-24（右）所示。

图中直观地对比了 Sigmoid 函数与其导数，可见无论 Sigmoid 的输入/输出是何值，Sigmoid 的导数都小于 0.25，这直接导致了梯度迅速下降、无法传递更深。

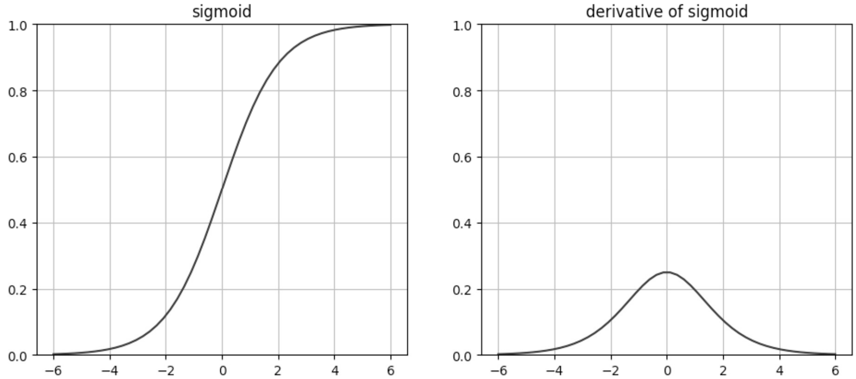


图 9-24　Sigmoid 函数与其导数

2. ReLU

找到了问题的原因，解决问题就变得非常容易：需要找到一个导数值固定为 1 的激活函数。Hahnloser 于 2000 年在《自然》杂志上发表了符合这个要求的激活函数——ReLU（Rectified Linear Units），函数形式非常简单：

$$f(x) = \max(x, 0)$$

也就是说，当 x 大于零时 $f(x) = x$，否则 $f(x) = 0$。把该函数用作激活函数有以下特点。

◎　必要条件：函数在 0 点形成拐点，因此函数显然是非线性的，满足激活函数的基本要求。

◎　优点：在 x 大于 0 时导数固定为 1，解决由激活函数导致的梯度消失问题。

◎　缺点：在 x 小于 0 时导数固定为 0，使得梯度传递中止。

其中第二点是当前 ReLU 被广泛应用的原因，而第三点则使得某些神经元变成所谓“dead ReLU”无法被训练。

因此，后来产生了一些 ReLU 的变种解决“dead ReLU”问题。

◎　Leaky ReLU：$f(x) = \begin{cases} x, \text{if } x > 0 \\ 0.01x, \text{otherwise} \end{cases}$。

◎　ELUs：$f(x) = \begin{cases} x, \text{if } x > 0 \\ a(e^x - 1), \text{otherwise} \end{cases}$，其中 a 是超参数。

以上形式都使得在遇到负值输入时导数不至于为零，解决了梯度中止的问题。

注意：由于“dead ReLU”和“开放上限”的原因，ReLU 通常只被用于卷积层，而深度学习网络中的全连接隐藏层仍然使用 Sigmoid。

3. Softmax

Softmax 是和 ReLU、Sigmoid 完全不同的一种激活函数，它总是出现在输出层，作用是给输出值做归一化处理。

所谓的归一化简单说来就是使得所有输出层神经元的激活值之和等于 1。Softmax 在计算中需要读取所有该层中的输入数据，该层中第 i 个神经元的计算公式为：

$$f(x_i)=\frac{e^{x_i}}{\sum_k e^{x_k}}$$

其中 k 是该层神经元总数，公式的形式保证了所有神经元的输出总和为 1。因此 Softmax 值常被用来直接解释为对应输出的“发生概率”。

9.3.6　Inception 与 ResNet

至此已经学习了卷积神经网络的所有基本要素，这里用当下最新的 CNN 网络架构完成 CNN 理论部分的学习。

1. Inception

第一代 Inception 网络结构如图 9-21 所示，不深究其中细节，从图中可以发现该网络由若干“总→分→总”的并行块串接而成（在图 9-21 的 Inception v1 中共有 9 个这样的并行块）。在谷歌后来的论文中这些并行块被称为“Inception module”，图 9-25 是 Inception-v4 论文中给出的一种 Inception module。

该模块与普通卷积网络不同的是，它用多种不同配置的卷积层同时对上一层数据进行计算，然后组合这些卷积输出。根据每个卷积层参数不同，Inception 网络中用到多种类似的 Inception module，未来肯定也会有更多出现。

对 Inception moduel 的设计没有太多数学原理，但从直观上可以理解为用型号不同的放大镜在上一层中寻找不同的模式，然后综合这些结果供下一层处理。

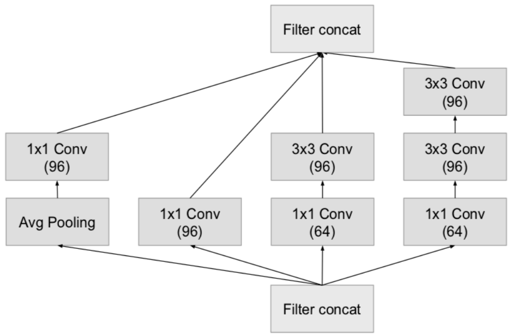


图 9-25　Inception module（取自 Szegedy 等人论文）

2. ResNet

ResNet（Residual Network，残差网络）是微软亚洲研究院赢得 2015 年 ILSVRC 竞赛的卷积网络模型。残差在数理统计中是指实际观察值与估计值之间的差值，在微软的 ResNet 中用来指代一层神经网络的输入与输出之间的差值。像 Inception module 是 Inception 网络的基因一样，所谓的 residual block 在 ResNet 中无处不在，图 9-26 是 residual block 的基本形态。

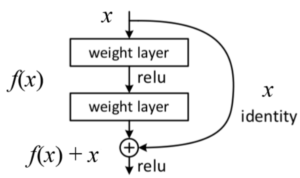


图 9-26　residual block 的基本形态（取自 Kaiming He 等人论文）

在图 9-26 所示的 residual block 上，输入向量 $\boldsymbol{x}$ 经过两个子隐藏层的计算用一个加法器将结果与原始输入 $\boldsymbol{x}$ 相加，将加法结果作为该 residual block 最终输出。其中的 $f(\boldsymbol{x})$ 就是所谓残差，子隐藏层学习的目的是该残差，而不是该层的最终输出。

设计这种带“短路”形态隐藏层的好处是：学习输出与输入近似的逻辑非常迅速。试想如果初始化该模块所有 weight 为 0，则该模块就是一个 $H(x)=x$ 的恒等运算。这样通过串联 residual block 可以一下子构建一个非常深（甚至几千层）的恒等 CNN 网络。之后在训练过程中只要微调各层参数即可实现复杂逻辑。

residual block 的提出在很大程度上解决了深度网络不可训练的问题，因此从 2015 年开始，“深度学习”的概念从原来的几十上百层提高到了上千层。

3. Inception-ResNet

谷歌的 Inception 提高了 CNN 的学习能力、微软的 ResNet 提高了网络深度和学习速度，如果将两者结合会产生什么结果呢？

在谷歌 2016 年的论文 *Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning* 中提出了 Inception-ResNet 网络概念，其核心模块是“带 ResNet 的 inception module”，如图 9-27 所示。

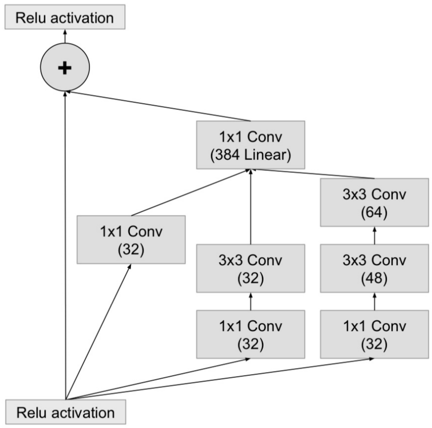


图 9-27　Inception-ResNet module（取自 Szegedy 等人论文）

论文中还公布了 Inception、ResNet、Inception-ResNet 在 ILSVRC 数据集上的实验结果，如图 9-28 所示。

Network	Crops	Top-1 Error	Top-5 Error
ResNet-151 [5]	dense	19.4%	4.5%
Inception-v3 [15]	144	18.9%	4.3%
Inception-ResNet-v1	144	18.8%	4.3%
Inception-v4	144	17.7%	3.8%
Inception-ResNet-v2	144	17.8%	3.7%

图 9-28　谷歌论文中对现代 CNN 的实验比较结果（取自 Szegedy 等人论文）

从该结果表中可以看出 Inception-ResNet(v2)与纯粹的 Inception(v4)网络相比识别效果几乎相同。那么加入 ResNet 基因的作用何在呢？

该论文同时给出了训练过程中不同 epoch 下各网络的 top-1 错误比较，如图 9-29 所示。

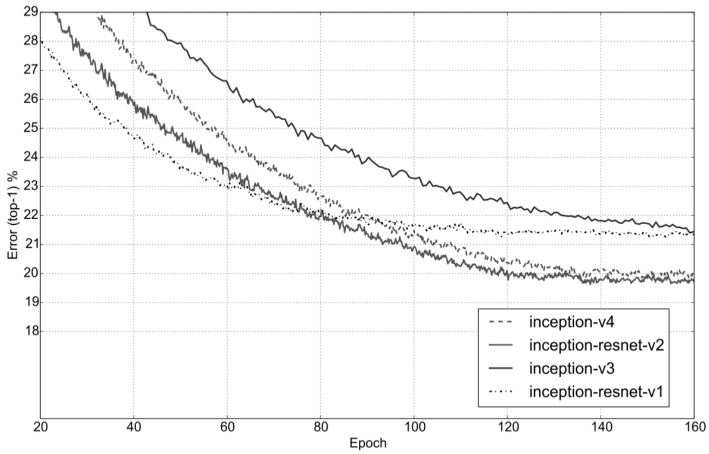


图 9-29　epoch 的 Inception、Inception-ResNet 效果比较（取自 Szegedy 等人论文）

从图 9-29 中可以看出，虽然 Inception-v4 与 Inception-ResNet-v2 最终都达到了相同级别的错误率，但 Inception-ResNet-v2 收敛得更快，比 Inception-v4 在更早的 epoch 达到该错误水平。该实验似乎进一步表达了“ResNet 使训练更快、Inception 使效果更好”的观点。

9.4 优化

在本章前文的学习中说明了神经网络使用随机梯度下降（SGD）算法进行反向传播优化训练。其实以 SGD 为基础还派生了很多其他优化算法，TensorFlow 等深度学习软件包对它们都有良好封装。本节学习特征规范化（normalization）、剪枝（drop out）在优化（optimization）过程中的重要作用，并简要介绍 Momentum 和 Adaptive 两类派生算法。

9.4.1 批次规范化

从 2015 年开始，谷歌在所有深度学习网络中加入了所谓的 Batch Norm（Batch Normalization）技术。该技术通过规范化（normalization）隐藏层的激活值达到加快优化速度的目的。

1. 什么是规范化

在传统的机器学习理论中规范化是样本数据预处理的一个步骤，该项处理有如下两个

目的。

◎　将不同定义域的数值特征统一映射到某个区间，通常是 0～1 之间。

◎　将样本数据在各维度特征上尽量形成“两边低，中间高”的正态分布。

比如用二维特征（年龄、收入）表示一个银行客户，年龄的取值范围是 0～150，而收入的取值范围是从 0 到非常大（比如 1 000 000）。对一批这样的样本数据做预处理则需要将它们都映射到 0～1 之间。最简单的规范化方法是大小值法，即对所有样本进行如下换算：

$$f(x)=\frac{x-\text{MIN}}{\text{MAX}-\text{MIN}}$$

其中 MAX、MIN 是该维度所有样本中的最大值和最小值。该函数的计算结果必定符合 0~1 的区间要求。此外，基于标准差的其他规范化方法能够满足特征正态分布化的要求。

2. 为什么要规范化

规范化的需求实际上来自大多数的机器学习模型都使用欧式距离衡量样本间的关系。如果不做规范化，算法会倾向于去学习取值范围大的维度。

注意：关于不同的距离衡量方法可参考本书第 4 章。

以神经网络使用的梯度下降训练算法来说，如果不做规范化将使取值范围大的神经元获得绝对值更大的导数（即梯度），使得每次迭代都在该神经元上做更多努力，反而忽略了其他同层神经元。

从数学上讲，这样的样本数据形成了非均匀的 Hessian 矩阵（即二阶导数矩阵），直观上的理解就是梯度过程会多走弯路，甚至迷路。

3. Batch Norm

在深度学习中，数据从输入层开始逐层传递，直到输出层。在数据预处理时执行的规范化传统只能保证输入层的数据有良好的规范化特性，但经过权值、偏置、激活函数的一系列计算后无法保证每个隐藏层都获得规范化的数据，使得隐藏层优化变慢。

Batch Norm 就是这样一种技术。在将数据传入隐藏层之前对数据进行规范化，保证隐藏层也能获得规范化的数据，达到提高整个网络优化速度的目的。

在谷歌论文 *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* 中给出了 Batch Norm 的具体实施算法，如图 9-30 所示。

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

图 9-30　Batch Norm 的具体实施算法（取自 Google 论文）

在该算法中，首先计算该批次数据的均值、方差（第一、二步），然后用方差作为分母对所有数据进行正太分布化（第三步），然后将取值范围约束在 0～1 之间（第四步）。需要特别指出的是第三步中的变量 ϵ 是一个随机数，以添加噪声来增强训练结果的鲁棒性。

9.4.2　剪枝

在 Batch Norm 出现之前，在深度学习网络中增强鲁棒性的工作主要是通过剪枝（dropout）技术完成的。现在该项技术仍被用在没有 Batch Norm 的隐藏层上。

1. 为什么需要剪枝

增强鲁棒性是为了让模型不对某个特征特别敏感、以使得预测过程完全依赖某个特征。为什么这样做呢？根本原因还是在于训练样本的有限性。比如一个识别水果类型的模型有如表 9-1 的样本集。

表 9-1　水果识别样本集

编号	特征			标签
	果肉 pH 值	大小	颜色	
1	7	小	绿	橘子
2	7	大	黄	苹果
3	6	小	黄	橘子

在该样本集中，所有的橘子都是“小”的，而苹果是“大”的。使用该样本集训练出的神经网络往往会产生这样的逻辑：不管其他特征如何，所有大的样本都是苹果，所有小的样本都是橘子。这个显然不是正确的逻辑，如何让神经网络在这种情况下还能兼顾所有特征呢？

神经网络的所有计算都通过神经元之间的数据传递完成，形象地将某些连接剪掉可以忽略该神经元的计算，如图 9-31 所示。

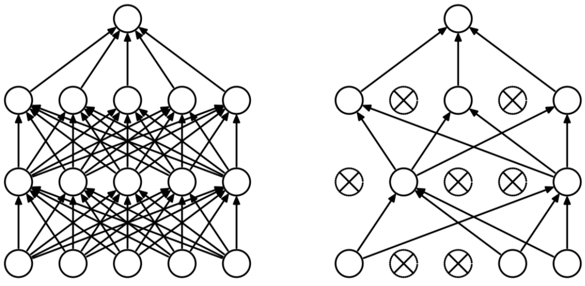


图 9-31　神经网络剪枝图例

（取自 Srivastava 等人论文 *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*）

图 9-31 的左侧是全连接网络，右侧是进行随机剪枝后的网络，该次剪切的作用是使得网络在该次训练中不得不忽略输入层的第二个、第三个神经元，还有一些其他隐藏单元。

如此在不同的训练迭代中随机剪切一些神经元，将使得网络不得不紧紧依靠其他神经元做出推理，即更加强壮。

2. 如何剪枝

通过图 9-31 固然可以解释剪切的动机与意图，但若在实际训练中每次随机搭建不同的剪枝网络也未免过于笨拙。

在当前的深度学习框架中，通过在隐藏层单元之前加入一个 dropout layer 可以达到剪枝的目的。可以将该 dropout layer 想象成一个随机开关，以某种概率阻断神经元激活值的传递。数学上该随机开关是一个 Bernoulli 分布，则 dropout layer 所做的工作就是：

$$f(x) = \text{Bernoulli}(p) \cdot x$$

其中 p 是 dropout layer 的超参数，用于配置剪切的力度。

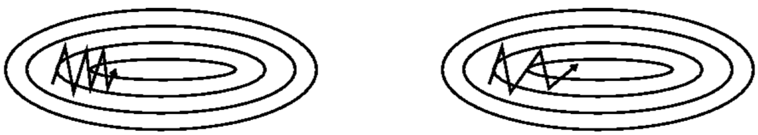
9.4.3 算法选择

在 TensorFlow 等深度学习框架中集成了很多派生于 SGD 的梯度下降优化算法，本节简要介绍它们的原理与算法。

1. Momentum SGD

如第 3 章所述，由于样本随机性的存在 SGD 的每次优化不一定是全局最优方向，这使得普通 SGD 不得不走很多弯路。

而 Momentum SGD 将梯度优化想象成一个在运动的点，在每次选择优化方向时除了考虑本次梯度值，还考虑之前的运动惯性方向。就像一辆高速行驶的车无法进行急转弯，加入动量的 SGD 也就减少了无用的“折返跑”，如图 9-32 所示。



（a）没有加入动量的 SGD （b）加入动量的 SGD

图 9-32 Momentum SGD（取自 Ruder 论文 *An overview of gradient descent optimization algorithms*）

自然地，每次选择方向在多大程度上考虑当前梯度，在多大程度上遵循已有惯性也就成为 Momentum 算法的一个超参数。

2. 自适应类算法

在 SGD 算法中有一个学习率超参数：learning rate，该值较大时梯度下降的步伐较大，反之则较小。而 Adagrad、Adadelta 等自适应优化算法的意图点是自动调节该参数，使得每次迭代、每个待搜索参数都有自己的 learning rate。

自适应调节 learning rate 是出于这样的理由：在训练刚开始时所有参数是随机值，此时需要较大的 learning rate 以找到大方向；此后随着迭代的进行应该逐渐降低学习率，以进行微调。同时，对梯度较大的参数使用较大的学习率可以加快该参数的优化。

这就是最先出现的 Adagrad 优化器的基本原理。随后出现的 Adadelta、RMSProp 等对该算法进行了调整，以避免学习率的单调下降无法跳出局部最优解。

而现在最流行的自适应类算法应该是 Adam，它结合了 Momentum 和 RMSProp 算法，

是一种带动量的自适应优化器。

3. 算法比较

现在放在深度学习开发者桌面的有 SGD、Momentum、Adadelta、Adam 等诸多优化器了。是不是 Adam 真的就是其在意图上所展现出的全能算法呢？

Adam 自 2014 年 12 月发布以来确实获得了越来越多的关注，但也有一些论文用实验对其产生了质疑。比如在 Berkeley 大学 Wilson 等人论文 *The Marginal Value of Adaptive Gradient Methods in Machine Learning* 中公布了不同优化器在著名图像库 CIFAR-10 上的分类实验结果，如图 9-33 所示。

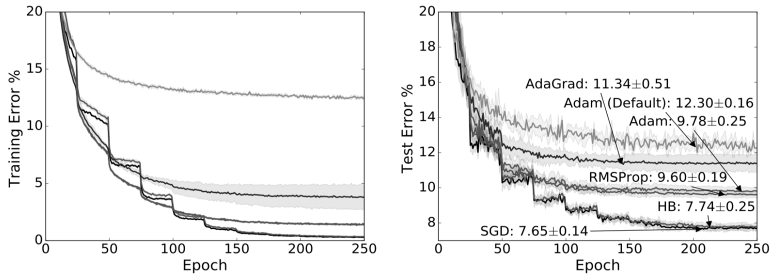


图 9-33　优化算法在 CIFAR-10 上的比较（取自 Wilson 等人论文）

从实验结果来看 Adam 达到的错误率并不理想，效果最好的反而是传统的 SGD；而 Adam 等自适应算法的优势仅体现在最开始的几个迭代上。

虽然某个具体的依赖于某数据集的实验不足以证明优化器的优劣，但这确实说明了对于数据研究者来说使用 Adam 不足以一劳永逸，重要的仍是用不同方法进行训练、验证、测试。

9.5　循环神经网络与递归神经网络

循环神经网络（RNN）是在 CNN 之外深度学习另一个主要应用领域。一些初学者可能会感到惊讶的是 RNN 同时是两种循环神经网络的名称：时间循环神经网络（Recurrent Neural Network）和结构循环神经网络（Recursive Neural Network）。由于前者的研究和应用更加活跃，在未说明的情况下“循环神经网络”一般被用于特指前者，而后者有另外一个中文名称：递归神经网络。本节先学习 Recurrent Neural Network 和其重要改进 LSTM，

然后介绍 Recursive Neural Network。

9.5.1 循环神经网络

如果用一句话来说明 RNN 存在的意义，那就是它是为处理时间序列特征数据而生的。传统神经网络（包括 CNN）和本书大多数其他模型所假设的模型输入都是静态数据，比如固定维度的客户特征、某种分辨率的图像等。而 RNN 适用于如下不能用固定数量维度进行特征建模的场景。

- ◎ 金融数据预测：无法确定一个交易日的价格与之前多少天的交易有关。
- ◎ 语音识别：输入语音有长句，也有短句。
- ◎ 双语机器翻译：类似语音识别，句子并非固定长度，文章并非固定篇幅。
- ◎ 智能输入法：根据已输入的若干汉字推荐接下来可能输入的词组。

在深度学习 RNN 广泛应用之前，由于计算能力与训练算法的限制解决这类问题的最有效方法是第 6 章学习的隐马尔可夫模型（HMM）。而现在基于 RNN 的时间序列预测效果已经超过 HMM。

1. 基本结构

与 HMM 假设时间点 t 的状态只与时间点 t-1 有关类似，RNN 也是一种以时间点为基础的建模工具。RNN 所假设的是时间点 t 能够接收来自时间点 t-1 的数据，同理 t-1 也能接收来自时间点 t-2 的数据。以此类推，时间点 t 的预测实际上取决于 t-1, t-2, t-3…这些历史上所有的输入数据，其示意图如图 9-34 所示。

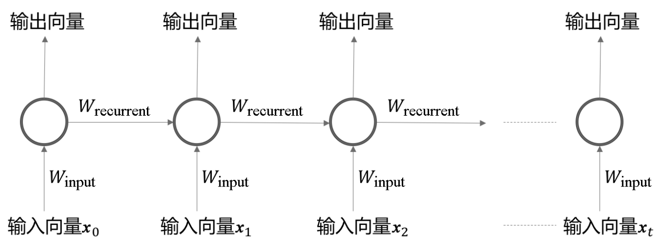


图 9-34　RNN 示意图

与本章之前的图例不同的是，图 9-34 的 RNN 中每个圆圈表达的不再是一个单独神经

元（cell），而是一层神经元（layer），因此其输入和输出都是一组向量。图中的 $\boldsymbol{W}_{\text{input}}$ 是之前普通神经网络中在层与层之间的权重向量，而 $\boldsymbol{W}_{\text{recurrent}}$ 是神经元接收上一个时间点同层输出（此时也称其为该层当前的状态）时使用的权重向量。

在图 9-34 中，同一层网络在不同时间点的权重向量 $\boldsymbol{W}_{\text{input}}$ 和 $\boldsymbol{W}_{\text{recurrent}}$ 都是同一组值，因此在实践中用如图 9-35 所示的网络就可以对 RNN 进行建模。

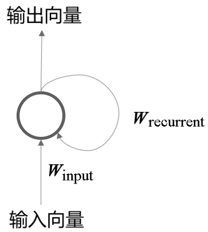


图 9-35　RNN 建模

如图 9-35 所示，RNN 模型在前向计算进行预测时的运行方式为：

◎　输入数据按时间序列逐个输入图 9-35 所示的网络中。

◎　在产生该时间点输出的同时，将该输出作为该层状态保留。

◎　在下一个时间点进行计算时，把保留的上一个时间点输出作为本次输入的一部分。

虽然 RNN 在结构上与普通神经网络的区别仅在于增加了“将本层输出作为下一个时间点本层输入”的回路，但在功能上达到了适应任意长度时间轴输入数据的目的。

2. 应用举例

以智能输入法的建模为例，为了使得每次输入的预测是基于之前所有输入的，可以将句子中的每个字/词组作为一个时间点的输入，获得如图 9-36 所示的 RNN。

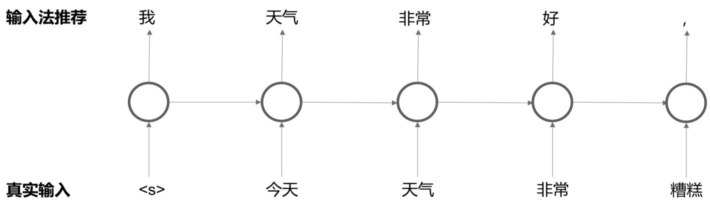


图 9-36　基于 RNN 的智能输入法

对图中结构解释如下。

◎ 在时间序列的第一个结点，为网络输入开始标识符（此处采用标签<s>，当然也可以用其他符号）。

◎ 对于开始符号<s>，网络只能给出最简单的预测，比如最常用的开始汉字“我”。

◎ 接着对网络逐个输入真实的词组：今天、天气、非常、糟糕。针对每一次输入，网络都给出它猜测的后续词组。

3. 更深的 RNN

既然 RNN 只是一种增加了本层回路的网络结构，它当然可以不仅是如前所述的单层网络，还可以将其设计成具有更深的结构，如图 9-37 所示。

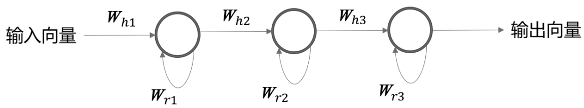


图 9-37 三个隐藏循环层的 RNN

图 9-37 网络是图 9-35 网络的直接扩展，只是增加了更多的隐藏层，而所有的循环回路都用于同层内部按时间轴传递信息。

4. RNN 的激活值与参数训练

经过前面的讨论，RNN 在本质上与普通神经网络并无太大差别，只是在每一层增加了一组输入向量：上一时间点本层的输出内容。因此 RNN 中一层神经元的激活值计算公式为：

$$A_t^l = f(A_t^{l-1}, S_{t-1}^l) = \sigma(W_h^l A_t^{l-1} + W_r^l A_{t-1}^l + b^l)$$

其中 A_t^l 是隐藏层 l 在时间点 t 的激活值，其取决于：

◎ 上一层在时间点 t 的激活值 A_t^{l-1}。

◎ 本层在时间点 t-1 的激活值/状态值 S_{t-1}^l。

◎ 本层偏置为 b^l。

有了 RNN 的激活值计算方法，就可以用本书 9.1.3 中学习的反向传播梯度下降法优化求解出最佳权重与偏置了。

具体过程仍然是对激活函数求导、权重、偏置的链式传播，此处不再细究。可以直接想到的是，由于 RNN 在普通网络之上又增加了按时间轴传递信息的机制，必定使得反向传播链更长、更复杂。由此而导致原本就存在的梯度消失、爆炸等问题变得更加严重，在解决这些问题之前 RNN 无法像想象中那样适应非常大的时间序列数据。

9.5.2　长短期记忆（LSTM）

RNN 在训练中存在的梯度消失与爆炸促使计算机科学家探索优化 RNN 的网络结构以解决这些问题，其中长短期记忆网络（Long Short-Term Memory，LSTM）是目前最流行的一种方案。

1. LSTM 意图

普通 RNN 用一组状态向量（即本层上一次的输出值）用于在时间点之前传递信息。该状态直接由上一个时间点的输出产生，该向量可以理解成一种“短期状态”。而 LSTM 的含义是它利用一个长期有效的短期记忆在时间轴上传递状态（该状态被称为 cell state），即该状态不仅传递给下一个时间结点，还能在之后的一段时间里长期有效。这是如何做到的呢？cell state 的传递逻辑如图 9-38 所示。



图 9-38　LSTM 的状态传递逻辑

图中用符号 C_t 表示在时间点 t 的单元状态（cell state）。它沿着时间轴在同层结点间传递信息，与普通 RNN 中短期状态不同的是，它在结点内部通过三个所谓的门（gate）进行了特殊处理以达到接收更早时间结点所产生信息的目的。

◎　遗忘门（forget gate）：结合本时间点的输入数据，从上一时间点给出的 cell state 中移除不需要的信息。

◎　输入门（input gate）：将本时间结点的输入数据集成到 cell state 中。

◎ 输出门（output gate）：计算本层激活值。

2. LSTM 结构

如果 LSTM 中的三个 gate 能够各自完成如上被定义的功能，则 cell state 就能更好地帮助每个结点进行输出预测。那么这些如魔术般起作用的 gate 是如何运行的呢？现在拿起“透视镜”，看看 LSTM 内部结构是什么样子的，如图 9-39 所示。

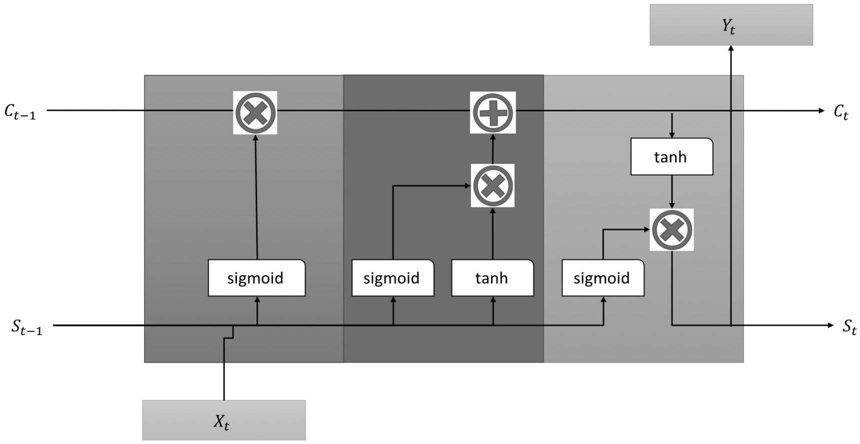


图 9-39 LSTM 内部结构

在图 9-39 中，C_t 仍然表示在时间点 t 的 cell state，而 S_t 是时间点 t 的隐藏值（被称为 hidden value）、$\boldsymbol{X}_t$ 是该层的输入向量，$\boldsymbol{Y}_t$ 是输出向量。

图中的 tanh 与 sigmoid 功能类似，也是一种激活函数。虽然没有标出，但与以往一样所有激活函数的输入其实都伴随着一组权值 W 与偏置 b，它们是 LSTM 中的待训练变量。

3. 如何忘记：乘法

如图 9-39 所示，LSTM 中所谓的 forget gate 就是将 (S_{t-1}, X_t) 激活运算的结果对 C_{t-1} 做乘法的操作。这为什么能起到“忘记”的作用呢？

这完全要归功于 sigmoid 函数。关于 sigmoid 一个众所周知的特性就是它的输出值域为 0～1 之间，因此将该结果乘以收到的前一个时间点的 cell state C_{t-1} 就可以起到将其削弱的目的，也就达到了忘记的效果。

在最极端的情况下：若遗忘门中 sigmoid 输出为 1，则 C_{t-1} 保持不变；若为 0，则 C_{t-1} 被清空。而该值具体为多少，完全取决于该激活函数的输入 (S_{t-1}, X_t)、权值、偏置，其中后两者是通过训练优化获得的，也就是说 LSTM 网络能学习到什么时候需要忘记 cell state、需要忘记多少 cell state。

4. 如何记住：加法

在学习了遗忘门的机制后，也就不难理解输入门了。在图 9-38 中赋予输入门的任务是“记住该记住的”。相对于遗忘门用乘法实现忘记，显而易见，记住的最佳方式是做加法。在图 9-39 中，(S_{t-1}, X_t) 经过两个激活函数的运算后将结果加入到了 cell state C_{t-1} 中，并确定了新的 cell state C_t。

与遗忘门类似，在输入门中记住多少信息完全取决于 (S_{t-1}, X_t)、权值、偏置。通过网络训练优化后两者的值，可以达到自动记忆的功能。

5. 输出门：综合 cell state、hidden value 与 *X*

在图 9-39 中，输出门（output gate）由两个激活函数构成，它们分别计算 C_t、(S_{t-1}, X_t) 的值，在将两者相乘后得到本层最终的输出 Y_t。同时该值也是本层新的 hidden value S_t。

至此，LSTM 完成了由输入 C_{t-1}, S_{t-1}, X_t 产生输出 C_t, S_t, Y_t 的工作流。

6. 为什么 LSTM 能解决 RNN 的梯度消失和爆炸

在 9.3.1 节中介绍了深度学习网络中梯度消失与爆炸的根本原因：梯度通过激活函数一阶导数进行链式传递，当激活函数一阶导数远小于 1 时产生梯度消失，当远大于 1 时产生梯度爆炸。

唯有激活函数一阶导数恒等于 1 时才能将梯度顺利进行反向传播，CNN 中目前主流的激活函数 ReLU 就是出于这一动机进行设计的。

而在 LSTM 中，cell state C 其实也使用了一阶导数等于 1 的恒等激活函数 $f(x) = x$。为什么这样说呢？

再次观察图 9-39，可以总结 *C* 的计算流程为公式 $C_t = \mathrm{forget} \times C_{t-1} + \mathrm{input}$。如果把遗忘门与输入门的值分别看成 *C* 的权值和偏执，其实该公式就是：

$$C_t = W \times C_{t-1} + b$$

这正是普通神经元的线性计算！而普通神经元产生梯度消失/爆炸的原因是，还要在该结果上再次进行激活函数计算。但 LSTM 的 C 根本就没有了外层激活函数，也可以认为它的激活函数是 $f(x)=x$。

因此，在 LSTM 中 cell state C 的传播是没有梯度下降或爆炸问题的，使其可以适用于时间轴非常深的 RNN 应用场景。

7. LSTM 的变形

实际上图 9-39 所介绍的只是应用最多、最容易被理解的 LSTM 结构之一。RNN 作为较活跃的研究领域派生了多种用于在时间轴上传递 cell state、hidden value 两种状态的 LSTM 结构。但是 LSTM 的精髓在于在 cell state 的计算中使用了恒等激活函数 $f(x)=x$，这些结构万变不离其宗，变化的也只是“三大门”内部的计算罢了。此处不再一一举例，在实践中如果遇到无须惊讶。

9.5.3 递归神经网络

递归（recursive）神经网络也是一种用于为不定特征维度数据进行建模的神经网络结构。但与循环神经网络为时间序列数据（list）建模不同的是，递归神经网络用于为树形数据（tree）建模。

1. 树形建模

最典型的应用是已知语法结构情况下的文本语义处理，比如对于文本：

```
The weather is good.
```

进行语法解析（parse）后可以做如下标记：

```
(S (NP the weather) (VP is) (ADJ good))
```

其中 S、NP、VP、ADJ 是常用的文法标记，分别对应如下含义。

◎ S：sentense，一句话。

◎ NP：noun phrase，名词短语。

◎ VP：verb phrase，动词短语。

◎　ADJ：adjctive，形容词。

如上的语法分析结果实际上是一个树形结构，如图 9-40 所示。

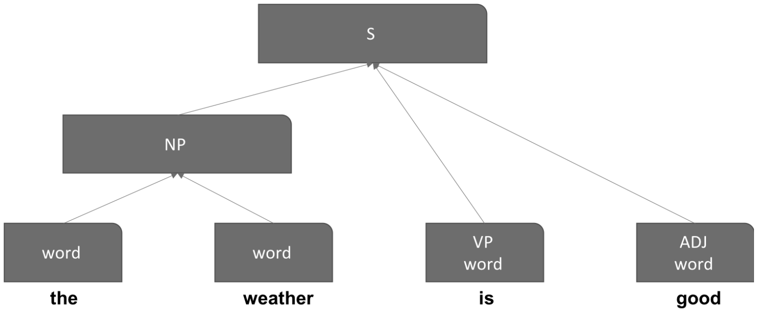


图 9-40　语法树树形结构举例

虽然这只是一段最简单的文本，但不难想象其实任意长度的文章或句子在进行语法解析后都可以形成类似的树结构。

2. 树形模型应用举例：文本情感分析

现在假设有这样一个任务：对任意长度的影评做情感分析，区分该段文本是喜欢、或不喜欢被评论的影片。用本书第 8 章学习的基本词袋模型很难取得满意效果，因为自然语言中充斥着很多“负负得正”的表达方式，很难通过否定词的数量统计进行情感判定，比如：

```
I like the movie.                                       # 喜欢
I don't like the movie.                                 # 不喜欢
I don't think anyone dislike this movie.                # 喜欢
My wife don't think anyone dislike this movie, but I don't agree.     #
不喜欢
```

处理这类问题不得不要求计算机逐级分解句子结构，获得每个单词、子句的情感含义并逐级汇总形成最终判断。如果用神经网络解析每一个元素的情感，则形成如图 9-41 所示的解析树。

在图 9-41 中每一个结点都是一个神经网络模型，用实心结点表示情感判断为“喜欢”，即肯定预测；用网线结点表达否定预测。结点上的文本“1→1”“2→1”等用于表示该神经网络模型的输入与输出神经元数量，即用于表达网络形状。可以看出，通过这样由下至上的分析很容易获得整体的最终结果。

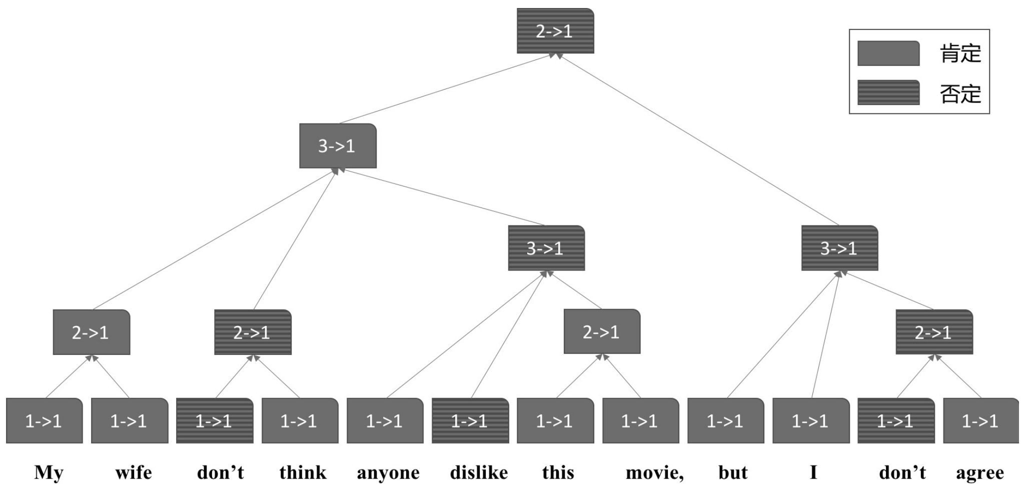


图 9-41 情感解析树

3. 递归神经网络解决树形问题

在图 9-41 所示的例子中，仅仅 12 个词的句子就用到了 20 个神经网络模型；当文本内容较多时，必定需要建立非常大的语法分析树才能完成上述任务。

当使用普通神经网络模型时，图 9-41 中的每一个树都需要单独训练，实际上不可能获得任意语句的所有需要的模型。但如果将图 9-41 的所有模型进行分类，会发现它们无外乎如下三种类型。

◎ 1→1 网络：判断一个单词的情感类型。

◎ 2→1 网络：判断两个单词/字句联合的情感类型。

◎ 3→1 网络：判断三个单词/字句联合的情感类型。

所以，只要建立有限的神经网络模型，在训练与预测时根据文本树结构动态为每个单词/字句选择合适的神经网络类型就可以了。

熟悉程序设计的读者应该知道这种逻辑就是一种递归调用，自然而然，这种由下至上、将某些子神经网络的输出重定向到另一些神经网络的输入、利用子神经网络逐步聚合完成某项任务的模型就被称为递归神经网络。

4. 递归网络与循环网络比较的讨论

递归与循环神经网络都是适用于不定特征维度的深度学习模型，其不同点在于所依赖的输入知识结构不同：递归网络处理树形模型，循环网络处理序列模型。由此产生的结果是：

◎ 对于处理相同问题来说，由于递归网络能利用特征之间的逻辑关系，预测效果往往好于循环网络。

◎ 递归网络在训练时只在树的层与层之间存在反向传播，而循环网络需要在所有结点上进行反向传播。假设递归网络用于处理二叉树结构，则平均深度只相当于循环网络的 $\log N / N$，使得递归网络更容易训练。

◎ 递归网络需要的输入条件过于结构化，在实际工业应用中很少有场景能够完全具备树形，因此循环网络反而比递归网络获得了更多的实际应用。

根据以上特点，递归网络的应用与学术研究目前都只处于不温不火的状况，其训练方法与普通网络的反向传播亦无太大差别，不再赘述其具体细节。

思考： RNN 是哪两种神经网络？

9.6 前沿精选

深度学习作为当前机器学习方面的终极工具，在各种场景有层出不穷的应用。本章介绍几种典型的网络结构，它们或利用 CNN、RNN 等深度学习基本手段实现综合功能，或启迪未来几年可能出现的新网络架构，供读者回味与思考。

这些模型有的没有开源实验数据、有的需要太大计算资源无法用个人计算机验证，为了保证内容的公允性，本节插图或数据图表多取自谷歌或其他科研机构发表的相关论文。

说明： 当前的机器学习已经到了比拼数据与计算能力的产业阶段，这也是近年来的创新均来自谷歌、微软等大科技公司而非象牙塔学术机构的原因。

9.6.1 物件检测模型

在图像处理领域，用单一 CNN 网络就能完成的图像识别/分类是一个较初级的问题，

一个更有难度的问题是如何实现图像中的物件检测（Object Detect）。所谓物件检测是从一幅大的图像中标注出感兴趣的物理对象的类型、并标注出具体位置，如图 9-42 所示。

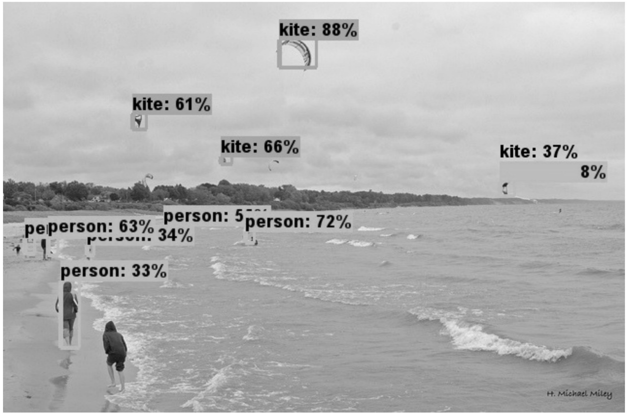


图 9-42　物件检测举例（取自谷歌论文）

注意：本小节插图均取自 2017 年谷歌论文 *Speed/accuracy trade-offs for modern convolutional object detectors*。

传统的物件检测一般通过三个步骤完成：

◎　利用图像的颜色等基础信息用滑动窗口提取若干个候选矩形区域。

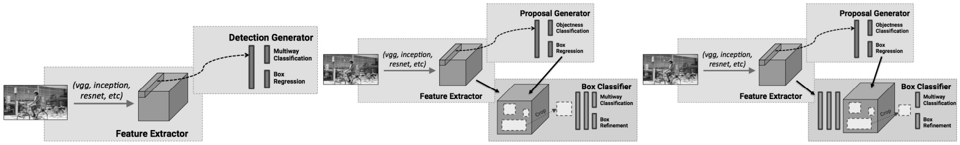
◎　用特定特征提取算法从每个候选区域中提取特征（比如对于人脸来说有著名的 Harr 算法）。

◎　将提取到的特征放入分类器检测该区域是否包含某目标物件。

由于卷积神经网络可以自动提取图像特征（回顾本章第 9.3.3 节），传统方法中第二步所需的各类特定算法已经完全没有必要存在，另外两步也可以通过深度学习工具加以改进。

1. 深度学习物件检测模型架构

随着深度学习模型尤其是卷积神经网络的成熟，计算机物件检测的精度与速度都得到了大幅度提高。其中比较有代表性的模型是 Faster R-CNN、SSD、R-FCN 等，这些模型的整体架构如图 9-43 所示。



(a) SSD.　(b) Faster RCNN.　(c) R-FCN.

图 9-43　三种物件检测模型架构（取自谷歌论文）

图 9-43 解读如下。

◎ SSD（Single Shot Detector）：用单一的卷积神经网络完成区域检测与目标分类。

◎ Faster R-CNN（Faster Region-based Convolutional Neural Networks）：先用一个卷积网络提取特征（feature map）并检测候选区域（proposals），然后用另一个网络合并特征与候选区域对每个区域进行分类（box classifier）。

◎ R-FCN（Region-based Fully Convolutional Networks）：与 R-CNN 类似，区别仅在于在特征（feature map）与候选区域（proposals）合并之前用全连接层进行降维处理，然后进行区域分类。降维操作的目的是使得后续分类计算速度更快。

以上模型均于 2016 年初左右被提出，其中 Faster R-CNN 之所以被冠以“Faster”是因为它是对 R-CNN（2013 年）、Fast R-CNN（2015 年）的改进。如今 Faster R-CNN 在性能与效果上均已超出其前身，使得 R-CNN 与 Fast R-CNN 已经不具有学习和应用的价值。

2. 性能与效果比较

深度学习物件检测模型只是利用到了卷积网络，并不限定 CNN 的具体结构，比如之前介绍的 VGG、Inception、RestNet 等都可以被放入 SSD、R-CNN、R-FCN 等物件检测模型中。因此在谷歌论文的实验环境中，分别实践各种 CNN 搭建的物件检测模型，并运行在相同数据集上进行时间计算、预测效果打分，如图 9-44 所示。

对该图解释如下。

◎ 图中横坐标是计算时间，纵坐标是检测效果评估（mAP：mean Average Precision）。

◎ 用三种点表示物件检测模型类型：圆形、方形、菱形分别代表 Faster R-CNN、R-FCN、SSD。

◎ 不同的颜色代表不同的 CNN 结构：Inception、ResetNet、VGG 等。

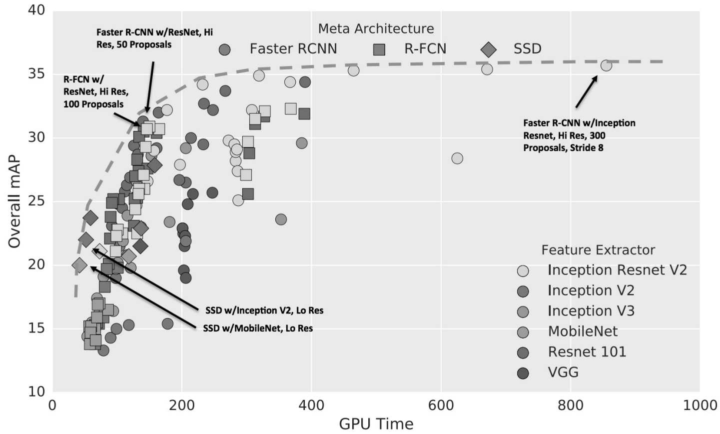


图 9-44　性能与检测评估打分（取自谷歌论文）

可以发现，SSD 在计算速度上有明显优势，而 Faster R-CNN 和 R-FCN 在预测效果上表现更好。另外，论文中还比较了它们在内存占用方面的不同，如图 9-45 所示。

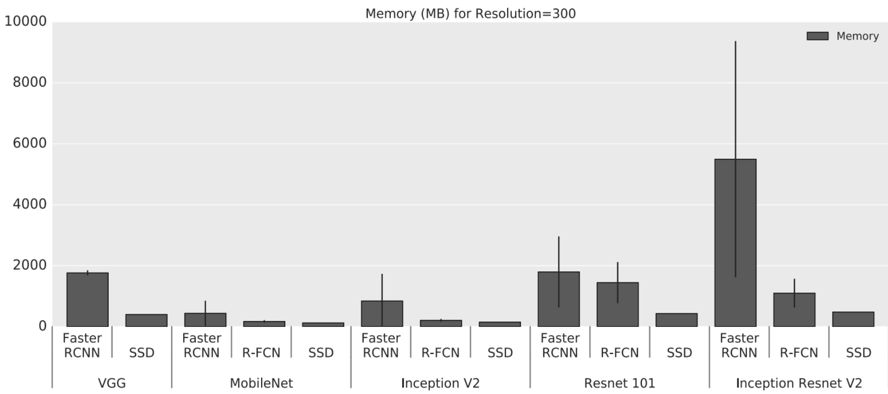


图 9-45　内存占用比较（取自谷歌论文）

图 9-45 的结论更加明显，在使用各种 CNN 的情况下，Faster R-CNN 都是占资源最多的模型，而 SSD 则最轻量级。

注意：图 9-45 中的直方图是平均内存占用，而其上的垂直竖线用于表示不用样本产生的内存占用方差。

3. 结论

在当前主流的物件检测模型中，SSD（包括与其架构类似的 YOLO，you only look once）

是对硬件资源要求最少的模型，在相同资源情况下检测速度最快；而 R-FCN 则在检测效果上提高很多；如果追求极致的检测准确度，应该使用 Faster R-CNN（尤其是以 Inception-Resnet-v2 作为 CNN 架构的 Faster R-CNN），但其对资源的要求最高。

9.6.2　密连卷积网络

通过 9.3 中对卷积基本方法的介绍，以及上一节中用不同 CNN 在物件检测中的实践对比，可以肯定当前 ResNet 的“短路”设计对增强 CNN 计算速度有较好效果。

相对于 ResNet，而密连卷积网络（Densely Connected Convolutional Networks，DenseNet）将短路设计发挥到了极致：将一层的神经元输出短路到后续所有层，加入它们的输入张量。这就是所谓的“Dense Block”，如图 9-46（左）所示。自然地，应用了 Dense Block 的神经网络被称为密连卷积网络。

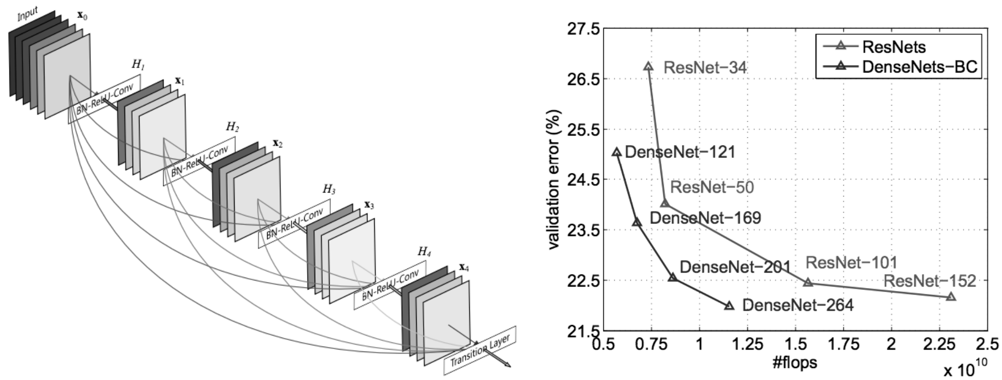


图 9-46　Dense Block（左）与实验数据（右）（取自论文 *Densely Connected Convolutional Networks*）

在论文中给出了 DenseNet 与 ReseNet 在 ImageNet 数据集上的效果对比，如图 9-46（右）所示，横坐标是所需计算资源（flop：每秒浮点运算次数），纵坐标是在 ImageNet 测试集上的 top–1 错误率，DenseNet-xxx、ResNet-xxx 等名称中的数字是网络规模。很明显，在使用类似计算资源的情况下，DenseNet 能获得远好于 ResNet 的分类效果。

DenseNet 被首次提出于 2016 年 8 月，巧合的是华中科技大学基础医学院的博士生导师韩芸耘作为第一作者于 2018 年 3 月 28 日在 *Nature* 上发表了关于生物大脑神经元信息传递规律的论文 *The logic of single-cell projections from visual cortex*。在其中打破了传统医学认为神经元间只进行简单映射的认知，发现实验小白鼠单个神经元里的信息可以有序地传递到多个完全不同的区域，如图 9-47 所示。

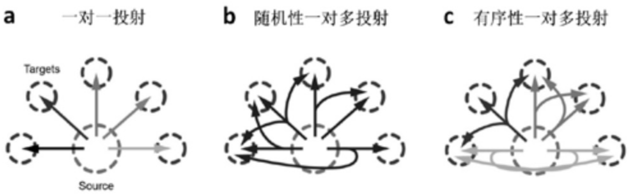


图 9-47　生物神经元映射（取自 *The logic of single-cell projections from visual cortex*）

两个学科交相呼应地发现了信息的一对多有序传递结构，这应该不是简单的巧合，也许这真的是自然界中一种高效信息传递的普适结构，人工神经网络在经过几十年的独立发展后似乎又与神经医学相聚在同一路口。

9.6.3　胶囊网络

从 2017 年底到 2018 年初，整个人工智能学术研究领域谈论最多的应该就是被誉为深度学习之父 Geoffrey E. Hinton 发表的论文 *Dynamic Routing Between Capsules*，其中介绍了全新的深度学习模型——胶囊网络（Capsule Network）。

1. 普通 CNN 的困境

虽然 CNN 在图像识别领域取得了巨大成功并掀起了深度学习浪潮，Hinton 指出 CNN 的工作方式与人类大脑大相径庭，继续沿着 CNN 的“卷积——池化”模型进行同构扩展无法达到更高的智能水平。

相对于普通神经网络，CNN 的主要创新在于用卷积层自动提取各层级的特征并用这些特征进行最终预测。然而卷积层使得神经元数大量增加，在提取特征后必须用池化层（一般为 Max-pooling）进行数据降维才形成最终特征，由此带来的问题是：

◎　Max-pooling 使得卷积层中的非局部最大点的信息丢失。

◎　Max-pooling 丢失了卷积识别出的特征在位置与角度等方面的信息。

第一条并非完全是一个缺点，它在某种程度上增加了网络的鲁棒性（参考本书 9.4.2）；但第二条会带来大问题，即它使得后续的处理无法识别特征之间的位置关系。

观察图 9-48（左），其中有四个几何特征：大矩形、小矩形、两个圆形，神经网络可以通过这四个特征学习到它是一辆汽车。但如果丢失了这些特征的位置信息，则很可能将同样具有这四个特征的图 9-48（右）也识别成一辆汽车。

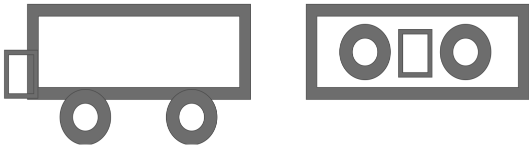


图 9-48　相同几何特征的不同位置组合成迥异的图形

2. 什么是胶囊

产生 CNN 困境的根本原因是在普通神经网络中执行运算的基本单位——神经元（neural）——处理的数据是标量（scalar），在卷积层后用抛弃神经元的方式降低数据复杂度直接丢失了很多信息。

而在新的架构中，承载运算的基本单元——胶囊（capsule）——处理的数据变成了向量（vector），用向量的线性变换降低数据复杂度使得所有信息都能参与其中。

这个改变推翻了本书 9.1.2 中讲述的沿用了几十年的基于神经元的网络计算方法，产生了全新的基于向量的胶囊运算流程。胶囊与传统神经元的运算方式对比如图 9-49 所示。

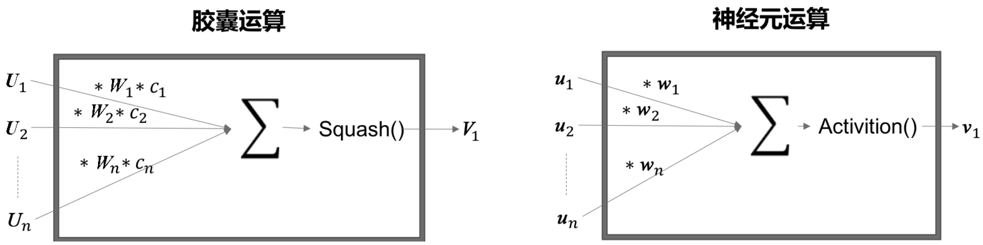


图 9-49　胶囊与传统神经元的运算方式对比

对该图解释如下：

- ◎ 胶囊的输入 $\boldsymbol{U}_1,\boldsymbol{U}_2\dots$ 与输出 $\boldsymbol{V}_1$ 均是向量，神经元输入与输出均是标量。
- ◎ 胶囊中的参数 $\boldsymbol{W}_x$ 是矩阵，用于对输入向量进行线性变换，从物理的角度理解就是做旋转、位移、缩放等处理，这部分功能是传统神经元中的缺失。
- ◎ 胶囊中的参数 c_x 是标量，作用相当于普通神经元中的 w_x，用于衡量每个输入的重要性。
- ◎ 普通神经元用各种 activation 函数做非线性变换，而神经元用所谓的 squash 函数对向量进行非线性变换。

在 Hinton 的论文中采用的 squash 函数形如 $s(\boldsymbol{X})=\frac{\|\boldsymbol{X}\|^2}{1+\|\boldsymbol{X}\|^2}\cdot\frac{\boldsymbol{X}}{\|\boldsymbol{X}\|}$，该函数保持向量 $\boldsymbol{X}$ 方向不变，而只改变向量 $\boldsymbol{X}$ 的长度。在胶囊网络中，向量的方向用于定义某种特征，而向量的长度用于表示该特征存在的可能性，因此可以将 squash 函数理解为对预测置信度的调整与归一化。

3. 网络架构

论文在提出了胶囊的概念后设计了一个用于识别 MNIST（数字手写图像库）的胶囊网络架构，如图 9-50 所示。

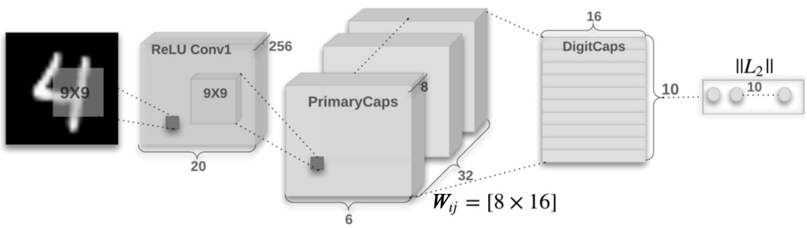


图 9-50　胶囊网络架构（取自 Hinton 论文）

对该网络解释如下：

◎　通过卷积层对原始图像进行特征提取，形成 256 个 feature map。

◎　与普通 CNN 不同的是，在卷积层后没有使用池化层进行数据降维，而是使用胶囊层 PrimaryCaps。

◎　由于卷积层有 256 个 feature map，因此胶囊层的每个输入向量是 256 维，即图 9-49 中的 $\boldsymbol{U}_1$、$\boldsymbol{U}_2$ 等都是 256 维向量。

◎　与卷积层每个神经元从原始图像中的某个局部区域扫描特征类似，每个胶囊也从上一层的某局部区域的向量获得输入，论文中每个 PrimaryCaps 中的胶囊有 9×9=81 个输入向量，即图 9-49 中的输入向量个数 n 等于 81。

◎　PrimaryCaps 中共有 6×6×32=1152 个胶囊，每个胶囊输出 8 维向量。

因此，卷积层中的标量数值共有 20×20×256=102400 个，而 PrimaryCaps 中的标量数值共有 1152×8=9216 个，达到了相当于池化层的降维目的，并且不会丢失卷积层中特征的位置信息。

在 PrimaryCaps 之后可以连接任意胶囊层继续搭建网络，其意义与在 CNN 中用多个卷积层逐层提炼更高级别知识的作用类似，因此不再解析图 9-50 的更多细节。

此外，论文中还提到了胶囊网络参数新的训练方法——动态路由算法（Dynamic Routing Algorithm），它用于替代传统网络中的反向传播对图 9-45 中的 $\boldsymbol{W}_x$ 参数进行训练，有兴趣的读者可以研读原论文。

9.7　CNN 实战：图像识别

在 TensorFlow 官网有一套完整的 CNN 图像识别实战教程，本节演示其应用步骤并解读其中关键代码。

9.7.1　开源图像库 CIFAR

对于初涉图像识别领域的开发者来说，想要实践机器学习的第一个问题往往是缺乏作为训练样本的图像数据库。其实仅从学习的角度来说，一些开源组织开放的图像库足够开发者训练自己的模型。

1. 图像库规模

本章 9.3 节中提到了 ILSVRC 竞赛数据库 ImageNet 就是最好的数据源之一，目前其中收纳了分布于 21 841 个标签的 14 197 122 张图片，每张图片的分辨率为 469x387 左右。对如此庞大的数据进行全量训练需要很长时间，并不适合初学者使用。

另一个体量较小的开源图像库是 CIFAR（Canadian Institute For Advanced Research，加拿大高等研究院），该图像库有两个版本。

◎　CIFAR-10：具有 10 个标签的数据，每个标签有 6000 个图像文件。

◎　CIFAR-100：具有 100 个标签的数据，每个标签有 600 个图像文件。

两个版本每张图片的分辨率均为 32x32。每个库的 60 000 个文件中，有 50 000 个是训练集图片，另 10 000 个是测试集图片。用深度学习模型在这样的数据规模上训练只需约几十分钟，本章案例使用 CNN 学习 CIFAR-10 版本图像库。

2. 如何获取

在 CIFAR 官网 https://www.cs.toronto.edu/~kriz/cifar.html 上直接提供了两个版本数据的 HTTP 下载链接，如图 9-51 所示。



图 9-51　CIFAR-10 下载页面

网站还给出了 CIFAR-10 数据样本举例，如图 9-52 所示。

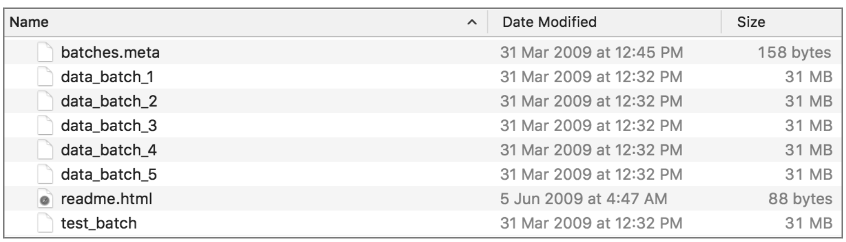


图 9-52　CIFAR-10 数据样本举例（取自 CIFAR 官网）

由图 9-52 可知，CIFAR-10 的 10 种标签为：airplane、automobile、bird、cat……

3. 读取 CIFAR 图像文件

从官网上下载的文件是一个.tar.gz 的压缩包，解压后的文件内容如图 9-53 所示。



Name	Date Modified	Size
batches.meta	31 Mar 2009 at 12:45 PM	158 bytes
data_batch_1	31 Mar 2009 at 12:32 PM	31 MB
data_batch_2	31 Mar 2009 at 12:32 PM	31 MB
data_batch_3	31 Mar 2009 at 12:32 PM	31 MB
data_batch_4	31 Mar 2009 at 12:32 PM	31 MB
data_batch_5	31 Mar 2009 at 12:32 PM	31 MB
readme.html	5 Jun 2009 at 4:47 AM	88 bytes
test_batch	31 Mar 2009 at 12:32 PM	31 MB

图 9-53　CIFAR-10 解压后的文件内容

CIFAR 图像库并非由简单的图像文件构成，而是一种 Python 对象文件（pickle）。如下代码演示了读取其中内容的方法：

```
import os
import pickle                                              # pickle 格式处理器
import numpy as np                                         # Numpy
import matplotlib.pyplot as plt                            # matplot
import random

def get_data(path, file):
    with open(os.path.join(path, file), 'rb') as fo:       # 打开 pickle 文件
        dict_obj = pickle.load(fo, encoding='bytes')       # 读取 pickle 对象
    X = np.asarray(dict_obj[b'data'].T).astype("uint8")# 图像文件数组
    Y = np.asarray(dict_obj[b'labels'])                    # 标签数组
    names = np.asarray(dict_obj[b'filenames'])             # 图像文件名数组
    return X,Y,names

def show_image(X,Y,names,id):                              # 用 matplot 显示某个文件
    rgb = X[:,id]
    img = rgb.reshape(3,32,32).transpose([1, 2, 0])
    plt.imshow(img)                                        # 绘制图像
    plt.title("%s with label %s"%(names[id], Y[id]))       # 显示文件名和 Label
    plt.show()

X, Y, names = get_data(data_dir, 'data_batch_1')
show_image(X,Y,names,random.randint(1,10000))  # 随机选取某个文件 id 并显示
```

代码执行结果如图 9-54 所示。

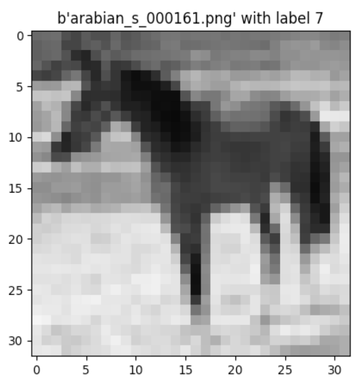


图 9-54　用 matplot 显示 CIFAR 图像

代码解释如下：

◎ get_data()函数演示了从 pickle 中读取图像数据、标签等信息的方法。

◎ show_image()函数演示了如何用 matplot 显示图像数据。

◎ 代码最后随机选取了一个图像 id 并显示。

由于 CIFAR 只是 32×32 分辨率的文件，所以多数图像实际显示起来非常模糊。虽然在之后的图像识别训练中无须显示某个具体图像，但上述代码有利于快速理解 CIFAR 数据与文件源格式。

9.7.2 项目介绍

TensorFlow 的示例教程维护在一个单独的 GitHub 项目中，可以通过 git 工具直接下载：

```
# git clone https://github.com/tensorflow/models.git
```

1. 项目组成

该项目包含了很多用 TensorFlow 搭建的深度学习模型，主要分为四大块。

◎ Official models：官方发布的针对当前 TensorFlow 版本经过较完备测试的模型，可拿来即用。

◎ Research models：是四大块中内容最多的一部分。由一些开发者提供，未必能在最新的 TensorFlow 版本中运行通过，建议只做设计参考。

◎ Samples folder：一些较小的组件，比如 Estimator 的使用、Java 语言集成等。

◎ Tutorial folder：官方发布的标准教程项目，由三部分组成：word2Vec、CNN 和 RNN。

在 Tutorial 中的三部分是当前深度学习应用最多的领域，其中 word2Vec 已经在第 8 章用 Gensim 实践过，本章将解析其中的一个 CNN 项目。

2. CIFAR-10 教程项目

在 tutorial folder 中，CIFAR-10 是最典型的 CNN 图像识别项目。进入如下目录，可以找到该项目的代码文件：

```
# cd models/tutorials/image/cifar10
```

```
# tree -L 1
|-- BUILD
|-- README.md
|-- __init__.py                                              // 初始化
|-- cifar10.py                                               // 模块核心
|-- cifar10_eval.py                                          // 评估接口
|-- cifar10_input.py                                         // 图像预处理
|-- cifar10_input_test.py
|-- cifar10_multi_gpu_train.py
`-- cifar10_train.py                                         // 训练接口
```

上述代码在注释中说明了关键文件的作用，其中 cifar10.py 是本项目的核心代码文件，其中包含了 TensorFlow 图的组建、优化器配置等。其他文件则为数据预处理或调用接口代码。

9.7.3　构建 Graph

在 cifar10_input.py 文件中包含了对 CIFAR-10 图像的预处理代码，之前已经对该数据源有详细介绍，因此跳过图像预处理部分直接分析 CNN 构建代码。

注意：本项目使用的 CIFAR-10 数据源格式与 9.1 节讲述的略有不同，有兴趣的读者可以细读 cifar10_input.py 源码。

在 cifar10.py 的 inference()函数中可以找到构建 CNN 模型的代码，该卷积网络架构类似图 9-19 所示的经典 LeNet 模型，由两组卷积/池化层后接两组全连接层组成。但是在网络中使用了 ReLU、normalization 等从 AlexNet 开始引入的“现代”特性。

1. 卷积、池化、规范化

如下是组建第一组卷积/池化层的代码：

```
def inference(images):
  ################# 省略若干注释 ##################
  with tf.variable_scope('conv1') as scope:
    # 初始化权重变量矩阵
    kernel = _variable_with_weight_decay('weights',
                                         shape=[5, 5, 3, 64],# 64 个 feature map
                                         stddev=5e-2,
                                         wd=None)
```

```
    #用conv2d定义卷积层
    conv = tf.nn.conv2d(images, kernel, [1, 1, 1, 1], padding='SAME')
    # 初始化64个偏置变量
    biases = _variable_on_cpu('biases', [64], tf.constant_initializer(0.0))
    pre_activation = tf.nn.bias_add(conv, biases)
    conv1 = tf.nn.relu(pre_activation, name=scope.name)# 卷积激活函数
    _activation_summary(conv1)                         # Tensorboard统计

  # 池化层
  pool1 = tf.nn.max_pool(conv1, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1],
               padding='SAME', name='pool1')

  # 规范化Normalization
  norm1 = tf.nn.lrn(pool1, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75,
               name='norm1')
```

解释如下。

◎ 代码中的_variable_with_weight_decay、_variable_on_cpu 等是在 cifar10.py 中的一些小工具函数，用于初始化 TensorFlow 待训练变量。

◎ 权重变量矩阵的形状为 shape=[5, 5, 3, 64]，含义为：

i．大小为 5x5 的卷积特征扫描窗口。

ii．上一层共 3 个平面，对应于输入图片的 R、G、B 三个颜色通道。

iii．本卷积层产生 64 个 feature map。

◎ tf.nn.conv2d 是 TensorFlow 二维卷积层类对象，类似的其他对象还有 conv1d、conv3d 等。

◎ 偏置变量的形状为 shape=[64]，对应于 64 个 feature map，初始值为 0。

◎ 在卷积层后用 tf.nn.max_pool 应用最大池化，ksize=[1, 3, 3, 1]用于定义 3x3 的池化窗口。

◎ 在池化层后用 tf.nn.lrn 进行输出规范化，LRN 的完整拼写是 local response normalization，该技术由 AlexNet 开始应用，该技术后来被 Batch Norm 取代。

在第一组卷积/池化层后定义了第二组卷积/池化层：

```
  with tf.variable_scope('conv2') as scope:
```

```
    kernel = _variable_with_weight_decay('weights',
                                   shape=[5, 5, 64, 64],# 上一层有 64 个平面
                                   stddev=5e-2,
                                   wd=None)
    conv = tf.nn.conv2d(norm1, kernel, [1, 1, 1, 1], padding='SAME')
    biases = _variable_on_cpu('biases', [64], tf.constant_initializer(0.1))
    pre_activation = tf.nn.bias_add(conv, biases)
    conv2 = tf.nn.relu(pre_activation, name=scope.name)
    _activation_summary(conv2)

  # norm2
  norm2 = tf.nn.lrn(conv2, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75,
                 name='norm2')
  # pool2
  pool2 = tf.nn.max_pool(norm2, ksize=[1, 3, 3, 1],
                 strides=[1, 2, 2, 1], padding='SAME', name='pool2')
```

第二组卷积层的构建与第一组基本相同，除了权重矩阵的形状改为了 shape=[5, 5, 64, 64]，这是因为从上一层中接收了 64 个 feature map 构成输入平面。此外，在第二组卷积后先进行 normalization，然后才进行池化，笔者认为这两层的先后顺序应无太大区别。

2. 全连接

接下来是两组全连接层的定义：

```
  with tf.variable_scope('local3') as scope:          #第一个全连接层
    # 接收 pool2 作为输入
    reshape = tf.reshape(pool2, [images.get_shape().as_list()[0], -1])
    dim = reshape.get_shape()[1].value
    weights = _variable_with_weight_decay('weights',
          shape=[dim, 384], # 384 个结点
          stddev=0.04, wd=0.004)
    biases = _variable_on_cpu('biases', [384],
          tf.constant_initializer (0.1))
    local3 = tf.nn.relu(tf.matmul(reshape, weights) + biases, # 激活函数
          name=scope.name)
    _activation_summary(local3)

  with tf.variable_scope('local4') as scope:          # 第二个全连接层
    weights = _variable_with_weight_decay('weights',
          shape=[384, 192], # 192 个结点
```

```
            stddev=0.04, wd=0.004)
      biases = _variable_on_cpu('biases', [192],
            tf.constant_initializer (0.1))
      local4 = tf.nn.relu(tf.matmul(local3, weights) + biases,
            name=scope. name)
      _activation_summary(local4)
```

上述代码并未使用 tf.layers.dense，而是用了更底层的接口创建全连接层：先定义权重矩阵、偏置矩阵，然后通过 tf.nn.relu(tf.matmul(reshape, weights) + biases, …)显示的定义全连接操作。

在权重矩阵定义中使用的 shape 参数是一个二元数组，其中第一个参数定义上一层结点个数，第二个参数定义本层结点数。在上述代码中，用 shape 参数定义了神经元数量分别为 384、192 的两个全连接层。

3. 输出层

最后是定义输出层，代码为：

```
    with tf.variable_scope('softmax_linear') as scope:
      weights = _variable_with_weight_decay('weights',
                          [192, NUM_CLASSES], # 形状
                           stddev=1/192.0, wd=None)
      biases = _variable_on_cpu('biases', [NUM_CLASSES],
                          tf.constant_initializer(0.0))
      softmax_linear = tf.add(tf.matmul(local4, weights), biases,
                          name=scope.name)  # softmax
      _activation_summary(softmax_linear)

    return softmax_linear                            # inference()函数返回
```

这里直接用一个由 NUM_CLASSES（在 CIFAR-10 中等于 10）神经元组成的全连接层作为 1-of-N 输出层，而并没有使用 Softmax，这是因为在后续训练的损失函数中应用了 Softmax 技术。

9.7.4 优化与训练

在定义模型的静态结构后，接下来分析本项目是如何调用会话进行模型训练的。从训练接口文件 cifar10_train.py 开始看起。

```
    def train():
      with tf.Graph().as_default():
    with tf.device('/cpu:0'):
         images, labels = cifar10.distorted_inputs()   # 获取样本数据

    logits = cifar10.inference(images)                 # 生成模型

        loss = cifar10.loss(logits, labels)            # 定义损失函数

    train_op = cifar10.train(loss, global_step)        # 优化计算

        with tf.train.MonitoredTrainingSession(…) as mon_sess:     # 建立会话，
（省略若干配置）
          while not mon_sess.should_stop():            # 循环迭代
            mon_sess.run(train_op)                     # 通过优化器训练
```

如果直接读源文件可能觉得有些杂乱。上面是从源文件 train()函数中删除了若干配置、注释代码后留下的关键代码。这样逻辑就非常清晰：从获取数据、建立模型、定义损失并训练、到最终迭代训练。

整个过程与 9.2 节中描述的 TensorFlow 开发流程完全一致，其中 inference()函数已经在上一节中解析过，现在回到 cifar10.py 看看解析损失与优化器代码：

```
    def loss(logits, labels):
      labels = tf.cast(labels, tf.int64)
      # 带 Softmax 的交叉熵
      cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
          labels=labels, logits=logits, name='cross_entropy_per_example')
      # 损失均值
      cross_entropy_mean = tf.reduce_mean(cross_entropy, name='cross_entropy')
      # 加入 Tensorboard 统计
      tf.add_to_collection('losses', cross_entropy_mean)

      return tf.add_n(tf.get_collection('losses'), name='total_loss')
```

上述代码的关键是用 tf.nn.sparse_softmax_cross_entropy_with_logits() 定义了作 softmax 激活后的真实标签（变量 labels）与预测标签（变量 logits）之间的交叉熵，并用 tf.reduce_mean()计算交叉熵的均值作为最终损失值。

交叉熵可以理解为一种衡量两组用张量表达的概率分布之间相似程度的标准。在数学上交叉熵 $H(p,q)=H(p)+D_{\mathrm{KL}}(p\parallel q)$，其中 $H(p)$ 是分布 p 的熵，$D_{\mathrm{KL}}(p\parallel q)$ 是之前学过的 Kullback–Leibler 散度。

接下来解析 cifar10.train()函数代码：

```
#从源代码中删减 TensorBoard 统计代码后的部分 train()函数代码
def train(total_loss, global_step):
  num_batches_per_epoch = NUM_EXAMPLES_PER_EPOCH_FOR_TRAIN /
                          FLAGS.batch_size    # 批次样本个数

  decay_steps = int(num_batches_per_epoch * NUM_EPOCHS_PER_DECAY)
  # 指数衰减 learning_rate
  lr = tf.train.exponential_decay(INITIAL_LEARNING_RATE,
                                  global_step,
                                  decay_steps,
                                  LEARNING_RATE_DECAY_FACTOR,
                                  staircase=True)


  with tf.control_dependencies([loss_averages_op]):
    opt = tf.train.GradientDescentOptimizer(lr)          # 梯度下降优化器
    grads = opt.compute_gradients(total_loss)            # 梯度计算

  # 封装梯度
  apply_gradient_op = opt.apply_gradients(grads, global_step=global_step)


  variable_averages = tf.train.ExponentialMovingAverage(
      MOVING_AVERAGE_DECAY, global_step)
  with tf.control_dependencies([apply_gradient_op]):
    variables_averages_op = variable_averages.apply(
        tf.trainable_variables())     # 所有可训练变量

  # 返回可以放入 Session.run()的待训练变量
  return variables_averages_op
```

其中关键点为：用 tf.train.exponential_decay()定义了指数下降的学习率，将其放入普通梯度下降优化器可以实现随着训练过程逐渐减小的 learning_rate，减少优化器在训练中的无用震荡。

注意：在 cifar10_eval.py 中实现了用 CIFAR-10 测试集评估已训练模型的功能，其中流程与训练类似，不再赘述。

9.7.5 运行

关键代码已经解析完毕，现在可以尝试启动 cifar10_train.py 开始模型训练，代码如下。

```
#cd models/tutorials/image/cifar10
#python3 cifar10_train.py

>> Downloading cifar-10-binary.tar.gz 100.0%
Successfully downloaded cifar-10-binary.tar.gz 170052171 bytes.
Filling queue with 20000 CIFAR images before starting to train. This will take a few minutes.
2018-05-14 10:38:01.621136: step 0, loss = 4.68 (408.8 examples/sec; 0.313 sec/batch)
2018-05-14 10:38:04.301559: step 10, loss = 4.65 (477.5 examples/sec; 0.268 sec/batch)
2018-05-14 10:38:06.901842: step 20, loss = 4.50 (492.3 examples/sec; 0.260 sec/batch)
…
```

如代码所示，在第一次运行 cifar10_train.py 时，程序会自动从 CIFAR 官网下载 CIFAR-10 图像库，下载完成后开始迭代训练，随着训练的进行 loss 值逐渐下降。

在运行训练程序约半个小时后，可以获得显著降低了的 loss 值。此时，可以另开一个控制台执行评估程序：

```
# python3 cifar10_eval.py
2018-05-14 13:42:47.194356: precision @ 1 = 0.852
2018-05-14 13:47:57.619786: precision @ 1 = 0.852
2018-05-14 13:53:08.019753: precision @ 1 = 0.851
..
```

该程序不会退出，而是每 5 分钟执行一次评估。在上述结果中，CIFAR-10 的图像识别已经达到 85%的正确率。

也可以打开 TensorBoard 查看整个训练过程的统计情况：

```
# python3 -m tensorboard.main --logdir /tmp/cifar10_train/
TensorBoard 1.9.0 at http://localhost:6006 (Press CTRL+C to quit)
```

注意：/tmp/cifar10_train/是模型训练时默认使用的日志目录，可以在执行训练时通过配置 train_dir 参数更改。

打开浏览器的 http://localhost:6006 地址，进入“SCALARS”可以查看各项统计，如图 9-55 所示。

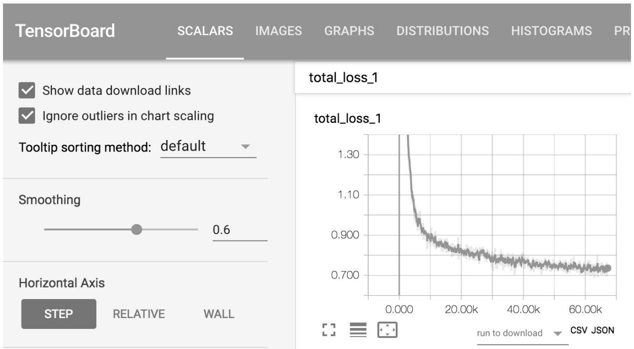


图 9-55　CIFAR-10 训练 loss 统计

进入“GRAPHS”可以得到整个图（Graph）的计算流程，如图 9-56 所示。

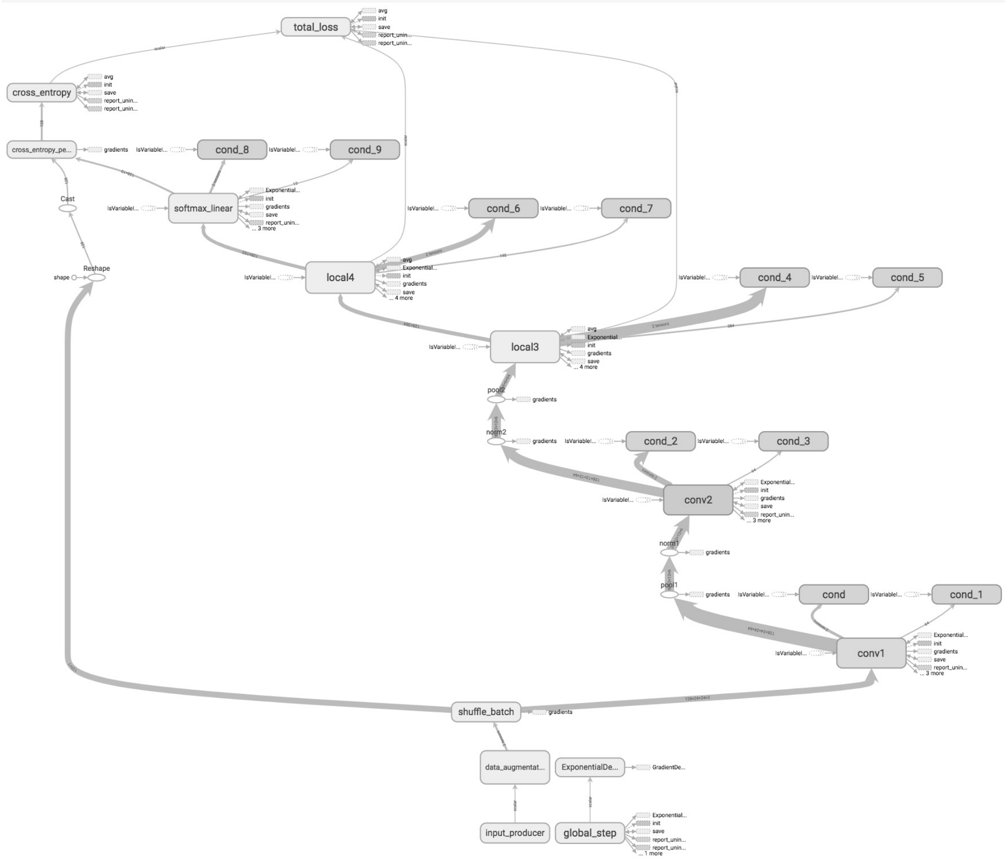


图 9-56　计算流程

在图 9-56 中从下到上可以看到计算全貌，图中从下到上：输入样本→shuffle→两次卷积与池化→两次全连接（local3、local4）→softmax_linear 输出层→计算交叉熵→计算损失值。

至此完成图像识别的案例解析，在本项目的 Official models 和 Research models 中还有很多其他更大的 CNN 模型，比如 official/resnet、research/inception 等，有兴趣自己搭建更复杂网络的读者可以参考这些模型。

9.8　RNN 实战：写诗机器人

语言模型是对人类自然语言的建模工具，有了它可以开发出很多“智能”应用，比如写诗机器人、对话机器人等。本节解析一个 TensorFlow 教程中的 RNN 实例，并将其改造为一个会写唐诗的机器人。

9.8.1　语言模型

语言模型是在语音识别、语义理解等领域出现的术语。具体而言，它是指对某种语言来说一组词汇组合所出现的概率分布 $P(w_1, w_2, \cdots, w_n)$，该分布对常见的单词组合（也可说是句子）能够获得较高的概率值，而对不常用句式或病句给出较低概率值。比如对于如下由英文单词组成的句子：

- ◎ Last year in the Philippines, earthquakes and tidal waves resulted in the deaths of more than 6,000 people.
- ◎ Tom's mother is a nurse at the hospital that's across the street from where Mary lives.
- ◎ Last that’s and tidal in the the where more at hospital.

所有学过英文或者以英文为母语的人应该都能理解前两句所表达的含义，而第三句的单词虽然也取自前两句内容，但却看不出该句子所要表达的意义。因此对于一个英文语言模型来说，前两句将获得较高的概率值，而第三句则会被判定为较低概率。

可以说是语言模型定义了世界上所有的语言，英语、法语、中文……或者 Java、Python 等。也可以认为所有人的脑中都有一个自己独特的语言模型，该模型形成了每个人的语言能力、说话习惯等。

说明：在自然语言处理领域，如果说不考虑词间顺序的词袋模型过于简陋，那么这里

所说的语言模型则几乎是终极武器。

1. 宾州树库（Penn Treebank）

对于人类来说，每个人脑中的语言模型并非与生俱来，而是由大脑从呱呱落地开始不断从家庭成员、学校环境中吸取培养而来。而如果想让机器人掌握一个语言模型，其实也要经历类似的过程，即阅读海量的语言资料，以掌握其中的规律。

语言资料可以看成一种单词序列数据，因此 RNN 也就成为了最适合的学习语言模型的机器学习工具。在 TensorFlow 官网的教程中，models/tutorials/rnn/ptb 就是一个学习语言模型的小型 RNN 项目（下文简称 ptb 项目），可以使用语言学领域著名的宾州树库（Penn Treebank）语料库训练该模型。

如下链接是一个宾州树库的子集，适合在个人电脑上训练语言模型：

```
http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz
```

在 Mac、Linux 电脑上可以通过如下命令下载并解压语料库：

```
# cd data-ptb                                             // 进入语料库存放目录
# wget http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz//下载
# tar -zxvf simple-examples.tgz                           // 解压
```

至此在目录 data-ptb/simple-examples/data/中生成了若干语料文件，在 ptb 项目中会用到其中的三个文件 ptb.train.txt、ptb.valid.txt、ptb.test.txt 分别作为训练、验证、测试集。对其中内容稍作预览，比如：

```
# head simple-examples/data/ptb.test.txt
no it was n't black monday
but while the new york stock exchange did n't fall apart friday as the dow jones industrial average plunged N points most of it in the final hour it barely managed to stay this side of chaos
some circuit breakers installed after the october N crash failed their first test traders say unable to cool the selling panic in both stocks and futures
…
```

可见语料库内容无它，只是简单的自然语言文本集。但是为了降低特征多样性、更适合用少量数据在个人电脑上训练，该语料库去除了标点符号和首字母大写等规则。

2. 训练模型

下载了语料库后即可尝试运行 ptb 项目训练语言模型：

```
# cd /models/tutorials/rnn/ptb              // 进入 ptb 项目目录
# python3 ptb_word_lm.py
  --model=small                             // RNN 规模，后续代码分析中讲解
  --data_path=/data-ptb/simple-examples/data // 语料库路径，读者实践时适当修改
  --num_gpus=0                              // 使用 CPU 训练
  --save_path=/tmp/ptb_model_example/       // 模型保存物理路径
```

注意：本节的 ptb 项目经过作者修改，与官方原始版本略有不同。请读者实践时从本书代码中寻找该项目，不要直接使用 GitHub 上的版本。

该命令会进入一个长期的训练状态，输出如下：

```
Epoch: 1 Learning rate: 1.000
0.004 perplexity: 7981.603 speed: 2940 wps
0.104 perplexity: 967.151 speed: 3211 wps
0.204 perplexity: 719.225 speed: 3021 wps
0.304 perplexity: 600.174 speed: 2859 wps
0.404 perplexity: 516.883 speed: 2886 wps
0.504 perplexity: 464.737 speed: 2941 wps
0.604 perplexity: 418.293 speed: 2970 wps
0.703 perplexity: 388.093 speed: 2984 wps
0.803 perplexity: 364.163 speed: 3014 wps
0.903 perplexity: 344.148 speed: 3014 wps
finished epoch
Epoch: 1 Train Perplexity: 325.944
finished epoch
Epoch: 1 Valid Perplexity: 218.544
Saving model to /tmp/ptb_model_example/.
Epoch: 2 Learning rate: 1.000
0.004 perplexity: 219.548 speed: 3237 wps
0.104 perplexity: 208.137 speed: 3286 wps
…
```

如上是完成一个 epoch 训练的结果日志，解释如下：

◎　第一列的 0.004、0.104 等是 epoch 内的训练进度。

◎　第二列的 perplexity 值是一种衡量语言模型成熟度的指标，模型与语料的匹配程

度越好 perplexity 的值越低。

◎ 第三列是该批样本训练速度。

每个 epoch 完成后会进行一次验证集的 perplexity 检验，该值不断降低，在--model=small 配置情况下经过 13 个 epoch 后训练结束。

3. 测试模型

读者可随时中断模型的训练过程，或者等待 13 个 epoch 完成，在这之后可以用如下命令查看模型在测试语料 ptb.test.txt 上的匹配程度：

```
# python3 ptb_word_lm.py
  --model=small                                  // 与训练时一致
  --data_path=/data-ptb/simple-examples/data // 语料库路径，读者实践时适当修改
  --num_gpus=0
  --save_path=/tmp/ptb_model_example/        // 与训练时一致
  --test_only=True
```

相比训练时，上述命令增加了参数--test_only=True，意为执行测试，输出如下：

```
Test Perplexity: 124.552
```

这是训练得到的模型对下载的原始 ptb.test.txt 文件的 perplexity 评分。现在编写如下 Python 代码以打乱原始测试集中的单词顺序：

```
def shuffle_file(file_name):
    shuffled = []
    with open(file_name) as f:
        for line in f:
            words = line.split()
            random.shuffle(words)
            shuffled.append(" ".join(words))

    with open(file_name, 'w') as f:
        for line in shuffled:
            f.write(line)
            f.write("\n")

# 根据文件地址适当修改
shuffle_file("data-ptb/simple-examples/data/ptb.test.txt")
```

执行上述代码后再预览文件内容：

```
# head data-ptb/simple-examples/data/ptb.test.txt
it was n't monday no black
fall while in barely friday jones the new stock managed stay average the points to of of but side dow it did york it this industrial apart as most plunged final n't N hour the chaos exchange
installed circuit breakers and in failed crash N both the futures panic first say some traders to october stocks selling the their unable test cool after
…
```

相比原始的文件内容，现在的文本中每一行仍保留了原来的所有单词，但由于打乱了单词顺序，已经很难读懂其中的含义。那么这样的文本能够得到语言模型什么样的 perplexity 评分呢？现在再次执行测试命令，可以得到如下结果：

```
Test Perplexity: 5481.113
```

和预料的一样，语言模型无法“读懂”这样的混乱文本，得到了非常高的困惑度（perplexity）。现在有理由相信，ptb 确实“学会”了英语。通过该模型已经可以开发出很多应用：任意文本的语言判别、作文语法打分等。

9.8.2　LSTM 开发步骤 1：网络架构

本书通过代码来解析 ptb 项目是如何建立和学习语言模型的。

该项目实际上是建立了一个如图 9-57 所示的 RNN 网络结构。其中每个单词通过词嵌入（embedding）技术从文本转换为词向量，然后被传入一个多层 RNN 网络（在 TensorFlow 中用 MultiRnnCell 类实现），最后用一个全连接网络（Dense）将输出转换为词向量。

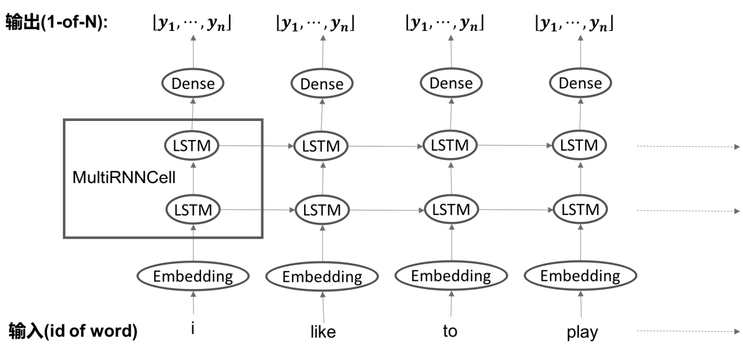


图 9-57　RNN 网络结构

输出层的每一个向量代表了根据之前的输入（不仅是当前时间点 t 的输入，还有 t–1、t–2...等历史输入）预测的下一个可能的单词。

在本网络中输入层与输出层都是对一个单词的编码，但编码方式略有不同。输入层使用的是单词的标量 ID，由网络通过 embedding 将 ID 转换为向量后进行计算；而输出层则使用 1-of-N 编码，便于后续通过 Softmax 等方法进行预测。

说明：对单词 ID、embedding、1-of-N 等概念不熟悉的读者请参考本书第 8 章。

9.8.3 LSTM 开发步骤 2：数据加载

ptb 项目的 Python 代码存放在三个源文件中：

```
#tree -L 1 git/models/tutorials/rnn/ptb/
|-- ptb_word_lm.py                                    // 主程序
|-- reader.py                                         // 样本数据加载
|-- util.py                                           // 工具函数库
```

主程序在开始执行后首先通过 reader.py 文件中的 ptb_raw_data()加载训练（train）、验证（valid）、测试（test）的所有数据。该函数主要内容解释如下：

```
def ptb_raw_data(data_path=None):          # data_path 是样本数据所在目录
  # 拼接出三个样本文件全路径名称
  train_path = os.path.join(data_path, "ptb.train.txt")
  valid_path = os.path.join(data_path, "ptb.valid.txt")
  test_path = os.path.join(data_path, "ptb.test.txt")

  word_to_id = _build_vocab(train_path)    # 建立“单词文本”→“ID”字典映射

  # 读取三个文本内容，并将所有单词转换为 ID
  train_data = _file_to_word_ids(train_path, word_to_id)
  valid_data = _file_to_word_ids(valid_path, word_to_id)
  test_data = _file_to_word_ids(test_path, word_to_id)

  vocabulary = len(word_to_id)             # 计算词汇总数
  words_by_id = [key for key in word_to_id]# 建立“ID”→“单词文本”列表映射

  return (train_data, valid_data, test_data, vocabulary, words_by_id,
         word_to_id)
```

该函数内使用的_build_vocab、_file_to_word_ids 等子函数都被写在 reader.py 文件中，有兴趣的读者可以进一步探究。

在最后该函数返回以 ID 号表示的所有文本内容、“单词文本”→“ID”映射、反向建立“ID”→“单词文本”映射。其中后两者对语言模型本身来说无用，但对于模型调用者来说可以用于构建其他输入，或将模型输出转换为文本。

9.8.4　LSTM 开发步骤 3：搭建 TensorFlow Graph

在加载样本数据后，主函数用三种样本分别传入 PTBModel 类、构建三个 PTBModel 对象，在该类初始化函数__init__()中搭建 graph，其中主要代码如下：

```
class PTBModel(object):

  def __init__(self, is_training, config, input_):
    #### 此处省略若干成员函数初始化代码 ###

    with tf.device("/cpu:0"):                     # 指定使用 CPU 进行词嵌入
      embedding = tf.get_variable(                # 用随机值初始化的词嵌入表
          "embedding", [vocab_size, size], dtype=data_type())
      self.embedding = embedding

    # 建立 RNN 网络
    output, state, outputs = self.build_rnn_graph(inputs,
                                                  config, is_training)

    softmax_w = tf.get_variable(                  # 定义全连接层 weight 变量
        "softmax_w", [size, vocab_size], dtype=data_type())
    softmax_b = tf.get_variable("softmax_b",      # 定义全连接层 bias 变量
     [vocab_size], dtype=data_type())
    # 即 logits= output*softmax_w+ softmax_b
    logits = tf.nn.xw_plus_b(output, softmax_w, softmax_b)
    logits = tf.reshape(logits,                   # reshape
             [self.batch_size, self.num_steps, vocab_size])
    self.logits = logits

### 省略损失计算代码，做后续讲解 ###
```

对比图 9-57 很容易理解如上代码，其内容可分为三块：

◎ 建立 shape=[vocab_size, size]、内容随机的张量，作为词嵌入查找表。

◎ 调用 build_rnn_graph()建立 RNN 网络，下一小节有详解。

◎ 定义 weight、bias 变量矩阵，并用 tf.nn.xw_plus_b()函数进行矩阵计算，这实际定义了一个没有激活函数的全连接层。

9.8.5 LSTM 开发步骤 4：解析 LSTM RNN

上述构建 Graph 的代码中，显然核心是 build_rnn_graph()函数，它使该 Graph 真正地成为了一个 RNN 网络。该函数经过一层封装后调用了_build_rnn_graph_lstm()，即搭建 LSTM 类型的 RNN，其代码如下：

```
def _build_rnn_graph_lstm(self, inputs, config, is_training):
    def make_cell():                                    # 建立一层 LSTM 单元
      cell = self._get_lstm_cell(config, is_training)
      if is_training and config.keep_prob < 1:
        cell = tf.contrib.rnn.DropoutWrapper(           # 网络 Dropout
            cell, output_keep_prob=config.keep_prob)
      return cell

    cell = tf.contrib.rnn.MultiRNNCell(                 # 多层 RNN 构造器
        [make_cell() for _ in range(config.num_layers)],
         state_is_tuple=True)

    # LSTM 初始状态
    self._initial_state = cell.zero_state(config.batch_size, data_type())
    state = self._initial_state
    self.compose_input = tf.placeholder(tf.int32,   # “单词 ID”形式的输入
            shape=[config.batch_size, self.num_steps],
            name="compose_input")
    compose_inputs = tf.nn.embedding_lookup(self.embedding, # 词嵌入
            self.compose_input)

    if self._is_training and config.keep_prob < 1: # 样本 Dropout
       compose_inputs = tf.nn.dropout(compose_inputs, config.keep_prob)

    outputs = []
    with tf.variable_scope("RNN"):
      for time_step in range(self.num_steps):         # 按“时间”展开 RNN
```

```
        if time_step > 0: tf.get_variable_scope().reuse_variables()
        (cell_output, state) = cell(            # 建立每个时间点的多层 RNN
            compose_inputs[:, time_step, :], state)
        outputs.append(cell_output)             # 保存输出
    output = tf.reshape(tf.concat(outputs, 1), [-1, config.hidden_size])
    return output, state, outputs
```

解释如下：

◎ 函数 make_cell()用于定义单个 LSTM，即构建图 9-57 中的一个 LSTM 块。

◎ 类 tf.contrib.rnn.MultiRNNCell()用于定义一个多层 LSTM，所有 LSTM 串行连接。图 9-57 中所示的是 2 层 LSTM。

◎ 代码中的 state 是 LSTM 的状态变量，即图 9-39 中的 *c* 状态；该变量初始值为 0。

◎ 函数 tf.nn.embedding_lookup()用于执行词嵌入，结果保存在 compose_inputs 张量中。

◎ 在训练时，用 tf.nn.dropout()对样本进行 dropout 处理，用 tf.contrib.rnn.DropoutWrapper()对网络中的连接作 dropout。

◎ 为了使 RNN 训练更高效，对其按时间点“展开”：在训练时为 num_steps 个时间点分别建立一套 MultiRNNCell。

◎ 倒数第四行中得到的（cell_output，state）分别对应于图 9-39 中的（*s* 状态，*c* 状态）。

注意：代码倒数第四行中，把 MultiRNNCell 类型变量 cell 当作函数进行调用。这是一个 Python 语法技巧，实际上其调用的是 MultiRNNCell.__call__()函数。

在函数 make_cell()中通过另一个子函数_get_lstm_cell()建立 LSTEM 单元，其内容为：

```
def _get_lstm_cell(self, config, is_training):
    if config.rnn_mode == BASIC:
      return tf.contrib.rnn.BasicLSTMCell(          # 建立普通 LSTM 单元
          config.hidden_size, forget_bias=0.0, state_is_tuple=True,
          reuse=not is_training)
    if config.rnn_mode == BLOCK:
      return tf.contrib.rnn.LSTMBlockCell(          # 建立 Block LSTM 单元
          config.hidden_size, forget_bias=0.0)
    raise ValueError("rnn_mode %s not supported" % config.rnn_mode)
```

即其通过配置的 LSTM 类型选择使用.BasicLSTMCell 建立普通 LSTM、或使用

LSTMBlockCell 建立 Block LSTM。其中 TensorFlow 的 Block LSTM 在速度上对 LSTM 进行了优化。

9.8.6 LSTM 开发步骤 5：LSTM 中的参数

之前的代码定义了整个网络的架构，但每一层单元的具体规模还取决于网络参数配置，比如代码中出现的 hidden_size、num_steps 等。在 ptb_word_lm.py 中用 XXConfig 对象定义了若干组参数，供使用者在训练/测试时选择。下面用 SmallConfig 举例，解释 LSTM 规模相关参数的含义：

```
class SmallConfig(object):
  num_layers = 2             # 在定义 MultiRNNCell 对象时使用的 LSTM 层数
  num_steps = 20             # 训练时一个 batch 传入多少时间点的数据
  hidden_size = 200          # LSTM 中隐藏状态向量的长度
  keep_prob = 1.0            # Dropout 保留的比例
  batch_size = 20            # 训练时一个 batch 传入多少个样本
  vocab_size = 10000         # 输出词向量的长度
  rnn_mode = BLOCK           # LSTM 类型
```

因此，每个 batch 传入输入层的数据是 num_steps×batch_size 个单词。

9.8.7 LSTM 开发步骤 6：用 sequence_loss 计算 RNN 损失值

至此已经分析完了网络的静态结构，现在回到 PTBModel.__init__()函数中看看 RNN 网络是如何定义损失计算策略的：

```
class PTBModel(object):
  def __init__(self, is_training, config, input_):
    ### 上接已分析过的网络结构 Graph ###

    self.targets = tf.placeholder(tf.int32,     # “正确”输出变量
        shape=[config.batch_size, self.num_steps], name="targets")
    # 用预测值 logits 与正确值 targets 计算损失
    loss = tf.contrib.seq2seq.sequence_loss(
        logits,
        self.targets,
        tf.ones([self.batch_size, self.num_steps], dtype=data_type()),
        average_across_timesteps=False,
```

```
        average_across_batch=True)

    self._cost = tf.reduce_sum(loss)            # 对所有损失值求和
    self._final_state = state
```

损失值的计算可分为三步：

◎ 用 placeholder 定义一个正确标签变量 self.targets，该变量用于接收训练样本中当前时间点下一个真实单词的 ID，因此该变量具有类型 tf.int32、形状=[config.batch_size, self.num_steps]。

◎ 用 tf.contrib.seq2seq.sequence_loss 比较预测张量 logits 与 self.targets，获得该 batch 内所有样本的损失值。注意此时 logits 是一个[config.batch_size, self.num_steps, vocab_size]形状的三阶张量，sequence_loss 的输出是一个[config.batch_size, self.num_steps]的二阶张量。

◎ 最后用 tf.reduce_sum()函数为损失张量中的所有元素求和，得到总体损失值（是一个标量）。

在 sequence_loss 内部，它先为传入的预测值 logits 作 soft_max 映射，然后计算与 targets 之间的交叉熵作为损失值，其结果是使得 logits 倾向于成为 1-of-N 编码的向量。

注意：sequence_loss 的具体算法在源码 tensorflow/contrib/seq2seq/python/ops/loss.py 中可以找到。

9.8.8　LSTM 开发步骤 7：学习速度可调优化器

在定义了损失值后，就可以配置以最小化该损失值作为目标的优化器了，代码仍旧在 PTBModel.__init__()函数中：

```
class PTBModel(object):
  def __init__(self, is_training, config, input_):
    ### 上接损失计算 Graph ###

    if not is_training:                         # 只在训练时加入后续优化计算
      return

    self._lr = tf.Variable(0.0, trainable=False)         # 学习率变量
    tvars = tf.trainable_variables()
```

```
        # 计算梯度
        grads, _ = tf.clip_by_global_norm(tf.gradients(self._cost, tvars),
                                          config.max_grad_norm)
        optimizer = tf.train.GradientDescentOptimizer(self._lr)# SGD 优化器
        self._train_op = optimizer.apply_gradients(            # 优化梯度
            zip(grads, tvars),
            global_step=tf.train.get_or_create_global_step())

        self._new_lr = tf.placeholder(                  # 学习率更新 placeholder
            tf.float32, shape=[], name="new_learning_rate")
        self._lr_update = tf.assign(self._lr, self._new_lr)
```

解释如下：

◎ 用函数 tf.gradients()计算损失值的梯度，并用 tf.clip_by_global_norm()防止梯度过大或过小（导致梯度爆炸或消失）。

◎ 使用 tf.train.GradientDescentOptimizer 优化器。

◎ 优化器的学习速度 self._lr 通过 placeholder 变量 self._new_lr 配置而来，使得训练过程中可以随时更改学习速度。

9.8.9 LSTM 开发步骤 8：训练

现在已经有了完整的网络定义与优化计算的 Graph，接下来的任务是应用 Session 执行该 Graph，以训练出语言模型。这部分代码定义在函数 ptb_word_lm.py 的 run_epoch 函数中，主要代码如下：

```
    def run_epoch(session, model, eval_op=None, verbose=False,
                  composing=False):
      state = session.run(model.initial_state) # 初始化 LSTM 第一个时间点状态
      fetches = {                              # 获取 Graph 中的变量计算结果
        "cost": model.cost,
        "final_state": model.final_state,
        "logits": model.logits,
      }

      for step in range(model.input.epoch_size):       # 每个迭代是一个 batch
        input_data = model.input.data[step*model.batch_size*model.num_steps:
                  (step+1)*model.batch_size*model.num_steps]
```

```
        input_data = np.reshape(input_data, [model.batch_size, -1])
        target_data= model.input.data[
                 step*model.batch_size*model.num_steps+1:
                 (step+1)*model.batch_size*model.num_steps+1]
        target_data = np.reshape(target_data, [model.batch_size, -1])
        feed_dict = {model.compose_input:input_data}    # 传入样本特征
        feed_dict[model.targets] = target_data          # 传入样本标签
        for i, (c, h) in enumerate(model.initial_state):
          feed_dict[c] = state[i].c                   # 传入上一个时间点的 LSTM 输出
          feed_dict[h] = state[i].h                   # 传入上一个时间点的 LSTM 状态

        vals = session.run(fetches, feed_dict)        # 执行 Graph
        cost = vals["cost"]                           # 读取结果
        state = vals["final_state"]
        logits = vals["logits"]

        costs += cost                                 # 累加损失值
        iters += model.input.num_steps

        # 打印该 batch 结果
        if verbose and step % (model.input.epoch_size // 10) == 10:
          print("%.3f perplexity: %.3f speed: %.0f wps" %
               (step * 1.0 / model.input.epoch_size, np.exp(costs / iters),
                iters * model.input.batch_size * max(1, FLAGS.num_gpus) /
                (time.time() - start_time)))

      return np.exp(costs / iters)                    # 返回整个 epoch 结果
```

函数使用的参数 Session 是在主函数中初始化的会话对象，model 是之前分析过的 PTBModel 对象。一个 epoch 由多个 batch 组成，是用所有样本对网络进行一次训练的过程。函数过程比较直接，即依次完成如下操作。

◎　初始化该 epoch 的 LSTM 状态。

◎　在字典 fetches 中定义每个 batch 执行图计算时需要读取的结果变量。

◎　逐个 batch 进行训练：

i．定义用于输入层的 placeholder 变量 compose_input 数据。

ii．定义用于损失计算的 placeholder 变量 targets 数据。

iii．传入上一时间点的 LSTM 状态。

iv．将以上参数传入 session.run()执行。

v．读取执行结果并打印。

其中 batch 总数 epoch_size 在样本加载时计算，值为 epoch_size=total_len/batch_size/num_steps。

9.8.10 开始写唐诗

ptb 项目建立的语言模型网络训练的唯一输入是语料文件 ptb.*.txt。在官网教程中使用英文宾州树库作为该文件语料来源，如果将语料库改为任何其他语言，就可以很容易地训练出其他语言模型。

语言模型不仅可以用于为不同语种建模，还可以表征特殊的语言习惯。可以想象，如果收集《唐诗三百首》中的唐诗作为语料库，即可训练出带有唐诗风格的语言模型。如果把模型本身的预测输出单词作为下一个时间点的输入单词，则可使模型完成“写作”任务，如图 9-58 所示。

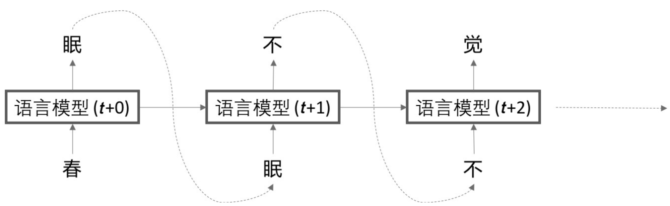


图 9-58 语言模型写作原理

通过图中的流程进行写作有一个缺点：给定相同的起始词，一定会产生相同的输出序列作为写作结果。因此在该过程中需要适当加入一些随机因素，使得写作内容更丰富。

9.8.11 写唐诗步骤 1：用唐诗语料训练语言模型

从很多网站可以下载到《唐诗三百首》语料库，作者下载了一份保存在本书代码路径 data-ptb/poet/tangshi.txt 中。通过如下简单代码可以从中提取出所有五言绝句或七言律诗：

```
def find_poet(line, k=5):                      # 从字符串 line 中提取出绝句或律诗
    chars = []
```

```
    for ch in line:
        if u'\u4e00' <= ch <= u'\u9fff':    # 只提取汉字编码
            chars.append(ch)
    # k 是单句长度，如果该行中有两句，则认为是唐诗
    if len(chars)==k*2:
        return " <s> " + " ".join(chars[:k])+" , " +
                " ".join(chars[k:])+ " "
    return None

def gen_poet_library(src_file, dest_dir, k=5):
    library = []
    with open(src_file) as f:                 # 打开原始《唐诗三百首》文件
        for line in f:
            poet = find_poet(line, k)          # 从每一行中提取唐诗
            if poet:
                library.append(poet)
                print(poet)
    total = len(library)
    # 保存结果文件
    with open(os.path.join(dest_dir, "ptb.train.txt"), 'w') as f:
        for line in library[:total*10//10]:
            f.write(line)
            f.write("\n")

    print("total %d sentences"%total)

gen_poet_library("data-ptb/poet/tangshi.txt",  # 原始文件名
               "data-ptb/poet",                  # 输出文件路径
               5)                                # 生成 k=5 的绝句唐诗
```

以上代码可以在目标文件夹内生成如下内容的 ptb.train.txt 语料库文件：

```
# head data-ptb/poet/ptb.train.txt
 <s> 垂 绥 饮 清 露 , 流 响 出 疏 桐
 <s> 居 高 声 自 远 , 非 是 藉 秋 风
 <s> 逐 舞 飘 轻 袖 , 传 歌 共 绕 梁
 <s> 动 枝 生 乱 影 , 吹 花 送 远 香
 …
```

其中每个词以空格分隔，<s>用作语句起始标识符。用类似的方法可以生成 ptb 项目需要的 ptb.valid.txt 和 ptb.test.txt 文件，用这些文件作为参数执行如下命令即可训练唐诗语言模型：

```
# python3 git/ models/tutorials/rnn/ptb/ptb_word_lm.py
      --model=tiny
      --data_pathdata-ptb/poet
      --num_gpus=0
      --save_path=/tmp/ptb_model/
```

命令使用了更适合短句的 tiny 参数组。在训练中学习速度随着 epoch 进程不断下降，如果发现 perplexity 值不再显著降低，可以尝试中断训练并重新开启，以使用最大的学习速度。执行过程与之前训练宾州树库时类似，此处不再重复列举。

9.8.12　写唐诗步骤 2：作诗

已经有了唐诗语言模型，接下来只要改造 run_epoch()函数使其按照图 9-58 所示流程执行 Session，就可以达到写诗目的了，关键代码如下：

```
def run_epoch(session, model, eval_op=None, verbose=False,
            composing=False):
  # 数据加载时产生的单词“文本”<->“ID”映射
  global words_by_id, word_to_id
  fetches = {
    "cost": model.cost,
    "final_state": model.final_state,
    "logits": model.logits,
  }

  composed = "<s> 春 来 雨 纷 纷".split(" ")      # 新诗的固定头部
  init_word_id = [word_to_id.get(word, 0) for word in composed]
  for step in range(composing and 1000 or model.input.epoch_size):
    input_data = model.input.data[step*model.batch_size*model.num_steps:
                      (step+1)*model.batch_size*model.num_steps]
    input_data = np.reshape(input_data, [model.batch_size, -1])
    target_data= model.input.data[
                      step*model.batch_size*model.num_steps+1:
                      (step+1)*model.batch_size*model.num_steps+1]
    target_data = np.reshape(target_data, [model.batch_size, -1])
    if composing:                                # 写诗模式
      if step >= len(init_word_id):              # 是否固定头部已输入
        # 在候选词中随机选择一个
        selected = random.randint(0, len(indices)-1)
        composed.append(words_by_id[indices[selected]])
```

```
            # 写入上一个时间点的输出
            feed_dict = {model.compose_input:[[indices[selected]]]}
          else:
            # 写入固定头部
            feed_dict = {model.compose_input:[init_word_id[step:step+1]]}
        else:
          feed_dict = {model.compose_input:input_data}
        feed_dict[model.targets] = target_data
        for i, (c, h) in enumerate(model.initial_state):
          feed_dict[c] = state[i].c
          feed_dict[h] = state[i].h

        vals = session.run(fetches, feed_dict)      # 执行计算
        logits = vals["logits"]                     # 读取“新写的”词向量

        if composing:
          indices, words = resolve_logits(logits[0][0]) # 将词向量转换为文本

      print("finished epoch ")
      if composing:                                 # 写诗完成，打印作品
        for word in composed:
          if word=='<eos>':                         # 过滤换行占位符
            print("")
          elif word=='<s>':                         # 过滤起始占位符
            continue
          else:
            print(word, end='')                     # 打印词
```

上述代码中的注释内容详细解释了程序流程，其中用到的一个子函数 resolve_logits() 用于将预测的词向量转换为词 ID 和文本：

```
    def softmax(x):
        return np.exp(x) / np.sum(np.exp(x), axis=0)

    def resolve_logits(logits, top_n=5):
      global words_by_id
      logits = softmax(logits[:len(words_by_id)])          # Softmax
      # 取出最有可能的 top_n 个预测单词 ID
      indices = logits.argsort()[::-1][:top_n]
      if logits[indices[0]]>0.05:
        # 去除 soft_max 值小于 0.05 的预测
        indices = [x for x in indices if logits[x]>0.05]
```

```
    return indices, [words_by_id[i] for i in indices]     # 返回词 ID 和词文本
```

该函数先将词向量转换为 Softmax 形式，取 Softmax 值最高的 top_n 个索引作为被预测的词 ID，并取出 Softmax 过小的值。为单个词向量提取 top_n 个输出的意义在于可以在 run_epoch 函数中随机选取一个作为“写作”结果，提高作品差异化程度。

9.8.13 写唐诗步骤 3：作品举例

现在用作诗模式运行 ptb_word_lm.py，就可以执行上述 run_epoch()流程。在上述代码中为新诗设计了固定开头“春来雨纷纷”，得到的结果如下：

```
# python3 git/ models/tutorials/rnn/ptb/ptb_word_lm.py
      --model=tiny
      --data_pathdata-ptb/poet
      --num_gpus=0
      --save_path=/tmp/ptb_model/
      --compose=True                                    # 写诗模式

春来雨纷纷,江手州姑试
今春周武赠,不作过若何
长童骑黄牛,歌声振林樾
意欲捕轻袖,凌歌共绕梁
煮豆生武影,离日风其稀
居人思明里,天涯共此时
圆月湖水平,涵虚混耻期
江蒸高飞泽,形曲岳寒城
欲济无舟楫,端居静来城
送家把日,能知何林色
八魄上飞雨,言上自相怜
万丈离旗树,迢有摘中西
早意璋西袖,歌忘迢半花
岂知明镜里,形影自相怜
万丈红泉落,迢多杂鼓声
深人具鸡场,把酒话桑麻
待到重阳日,还来就菊花
游人具鸡场酒,醉我至田家
绿树村边合,青山郭外斜
开轩面见至,山中发相平
…
```

诗的长度取决于在 run_epoch()函数中执行的迭代数量，如此可以让模型写出无穷尽的“唐诗”。更改“春来雨纷纷”的固定头部，可以得到完全不同的写作结果。

由于样本语料过少（目前样本库有 235 行五言诗），导致学习过程中无法应用验证集监控何时停止训练，因此学到的模型出现对训练样本的过度拟合。在上述作品中，出现了“天涯共此时”“形影自相怜”“绿树村边合，青山郭外斜”等个别完全复制真实唐诗的情况，就是因为这一原因。

如需防止这种情况的出现，可以在 run_epoch()和 resolve_logits()函数中增大随机力度，或者更早中断训练过程，使模型 perplexity 值略高出当前模型。这些作为练习留给读者自行探索，可尝试的其他练习还有：用该模型训练和写作“七言绝句”，修改损失函数加入对韵律、对仗等的要求。

9.9　本章内容回顾

◎　人工神经网络是一种分层计算模型，通过反向传播的梯度下降实现网络参数的学习。

◎　网络神经元的基本计算模型是 $\sigma(\boldsymbol{aw}+b)$，其中 σ 是激活函数，$\boldsymbol{a}$ 是输入向量、$\boldsymbol{w}$ 是权重向量、b 是偏置。

◎　Sigmoid 是最普通的一种激活函数，其值域范围为 0～1。

◎　理论上两个隐藏层的神经网络足以拟合任意函数。

◎　张量（Tensor）是 TensorFlow 的数据元素，零阶张量即标量，一阶张量即向量，二阶张量即矩阵。

◎　评估器（estimator）是 TensorFlow 的一种高层接口，其使用方式类似于 scikit-learn 中的模型。

◎　TensorFlow 用图（Graph）定义计算流程，用会话（Session）执行该流程。

◎　使用 TensorBoard 可以对图和各种统计数据实现可视化。

◎　随着网络的加深，必须解决梯度下降与梯度爆炸问题才能训练出可用的网络。

◎　卷积网络实际是一种分层的知识学习架构，其中每个卷积层的作用是特征提取、

每个池化层的作用是特征降维，通常在网络输出层之前有 1～2 层全连接层用于基本回归或分类。

◎ ReLU 函数的导数值是常数 1，因此能够解决由激活函数导致的梯度下降与梯度爆炸问题。ReLU 的衍生激活函数 Leaky ReLU、ELUs 等解决了 ReLU 存在死区的问题。

◎ Softmax 函数用于对输出向量归一化，它使所有元素之和等于 1 的性质使得可以将输出向量解释为概率分布。

◎ Inception 与 ResNet 是源自谷歌和微软的两种被普遍认可的 CNN 网络架构，它们能使神经网络更宽、更深。

◎ 规范化（normalization）是使所有特征具有相同取值范围甚至相同分布的预处理技术，它使梯度下降算法更有可能找到最优解。

◎ 批次规范化（Batch Norm）是一种在网络内部对各层输入做规范化的技术。

◎ 剪枝（dropout）能增强网络模型的鲁棒性。

◎ 随机梯度下降（SGD）有很多衍生算法，比如 Momentom、RMSProp、Adam 等，这些算法使得算法收敛更快、减少人工干预，但最终效果不一定更好。

◎ RNN 一般指时间循环(recurrent)神经网络，它用于处理序列化的特征数据，LSTM 是当前被普遍采用的 RNN 单元，它提高了 RNN 的可训练性。

◎ 递归（recursive）神经网络用于处理树形特征数据。

◎ Faster R-CNN、SSD、YOLO、R-FCN 等是物件检测（object detect）模型，其中 Faster R-CNN 效果最好，SSD 和 YOLO 执行最快。

◎ 密连（density）网络是一种跃层连接设计。

◎ 胶囊（capsule）网络着眼于解决 CNN 丢失特征间位置信息、无法识别特征线性变换的问题。其原理是去除普通神经元以标量为基本运算单位的行为，代之以向量为基本运算单位。

◎ TensorFlow 在 GitHub 上的项目在 https://github.com/tensorflow/models.git 中有很多正式或非正式的深度网络模型，非常适合在建模时学习参考。

◎ 一些开源组织提供了训练图像与自然语言模型需要的标准样本数据，比如

CIFAR、ImageNet 的图像库，Penn Treebank 的语料库等。

◎ TensorFlow 对深度学习网络中的大多数通用模块提供了封装类，本章案例中用到的有 tf.nn.conv2d（二维卷积）、tf.nn.max_pool（最大池化）、tf.nn.lrn（规范化）、= tf.nn.sparse_softmax_cross_entropy_with_logits（先 Softmax 然后计算交叉熵）、tf.reduce_mean（元素均值）、tf.reduce_sum（元素和）、tf.nn.dropout（数据剪枝）、tf.contrib.rnn.BasicLSTMCell（单层 LSTM）、tf.contrib.rnn.MultiRNNCell（多层 LSTM）、tf.contrib.rnn.DropoutWrapper（RNN 网络剪枝）等，在开发使用过程中可以参考在线文档灵活运用。

10

第 10 章 强化学习

随着 DeepMind 公司结合深度学习与强化学习开发的机器人在围棋上打败人类最顶尖高手，强化学习（Reinforcement Learning）获得了越来越多的社会关注。深度学习是机器学习中略显与众不同的一个分支，有监督与无监督模型都假设已有样本数据集进而挖掘模型参数；而强化学习则是一边探索数据一边学习模型。本书介绍强化学习的一般应用、主要理论及开发案例，主要内容如下。

◎ 场景与应用：强化学习模型的应用场景、常用术语、基本算法（Sarsa、Policy Gradient、Actor-Critic）。

◎ OpenAI Gym：一个强化学习算法的实验环境。

◎ 深度强化学习：DeepMind 用深度学习模型改造传统强化学习产生的一类算法如 DQN、DPN、A3C 等。

◎ 博弈问题：人工智能对博弈问题建模的基本技巧与策略。

◎　案例：改造 AlphaGo Zero，开发一个能够自我学习的中国象棋博弈机器人。

10.1　场景与原理

强化学习也许是机器学习中最接近大众所设想的人工智能模样的领域，它假设学习可以通过在所处任务环境中不断尝试并吸取经验，最后成长为解决相关问题的专家。

10.1.1　借 AlphaGo 谈人工智能

棋类博弈是当前人工智能体现自身能力的主要途径，人类历史上最著名的两个博弈机器人里程碑是 1997 年 IBM 的 DeepBlue 打败最好的人类国际象棋手卡斯帕罗夫和 2015 年 DeepMind 的 AlphaGo 打败人类顶级围棋手李世石。

提示：DeepBlue 的命名是由于 IBM 在科技界被称为“蓝色巨人”；而 AlphaGo 显然是因为“围棋”在英文中的名称是“Go”。

1. DeepBlue 是人工智能吗

两者在各自的时代获得了相似的关注度，都引出了类似“智能机器人是否会取代人类”这样的社会话题，但两者背后依赖的技术却不太一样。

DeepBlue 的核心是一个知识库系统，其中蕴涵了几位顶级国际象棋专家总结的领域知识，通过该知识库可以评估出任意一个给定局面的优劣。有了该套知识系统，DeepBlue 可以在每一步棋通过穷举方法找出最有利的走法。

IBM 也承认 DeepBlue 的技术仅能适用于国际象棋任务，因此 DeepBlue 并非一个智能系统，而是被 IBM 用于证明其在当时领先于世的硬件计算能力。

2. 会学习的 AlphaGo

在随后的近十年间有人尝试将与 DeepBlue 类似的方法应用在围棋博弈中，然而因为围棋的复杂度远高于国际象棋而无法建立能评估任意局面的核心知识库。

AlphaGo 走了与 DeepBlue 不同的另一条路，它从已有的人类围棋实盘自行学习围棋技巧并最终打败人类，显然这是一种机器学习方法。

这也导致了两者不同的命运：DeepBlue 在和卡斯帕罗夫的比赛结束后马上宣布退役；AlphaGo 在打败李世石后被不断改进，DeepMind 在 2017 年发布的 AlphaGo Zero 已经能摆脱任何样本数据，并通过自我对弈获得更高的能力。

3. 强化学习动机

AlphaGo 的自我学习能力正是来自最接近人工智能的机器学习方法——强化学习。还记得自己是怎么学习下棋的吗？大多数人不是从阅读和记忆大量棋谱开始的，而是在不断的实战中积累经验：能够赢棋的走法会被重复使用，输棋的走法则会在之后尽量回避。这恰巧也是强化学习的训练过程，这样很容易理解第 1 章的图 1-12 中所描述的强化学习模型的五个基本要素。

◎ 代理（Agent，A）：强化学习的主体，比如下棋者本人。

◎ 环境（Environment，E）：Agent 所处的世界，能接收 Agent 输入并给予反馈的，比如下棋的规则和博弈对手都是 Agent 的环境。

◎ 行为（Action，A）：Agent 对 Environment 所做的一次输入，比如走一步棋。

◎ 状态（State，S）：Agent 在执行某个 Action 后能观测到的所有信息，对下棋来说就是当前棋盘每个棋子的位置。

◎ 反馈（Reward，R）：在某个 Action 后，Environment 给予 Agent 的奖励（正 Reward）或惩罚（负 Reward），比如吃了某些棋子、赢棋、输棋。

代理必须具有某种记忆功能，通过不断地向环境执行行为并记录收到的反馈来学习知识，在以后的行为中避免执行获得负反馈的行为。

虽然棋类博弈在数学逻辑上相对于人类的其他日常工作生活复杂得多，但它的所有 Action、State、Reward 都可以被明确定义。这也是人工智能屡次在棋类取得突破，但尚未在生活领域被普遍应用的原因。

4. 策略与价值

除了图 1-12 中所描述的强化学习的五个要素，策略与价值是学习具体的强化学习方法之前另外两个需要理解的概念。

◎ 策略（Policy，π）：用于描述在某种 State 下 Agent 如何决定执行何种 Action。

在当前的已有算法中主要用概率分布的形式描述策略。比如，五子棋的第一步棋的走法策略可以是“棋盘中心点概率最高，外围位置逐步递减”。

◎ 价值（Value，V）：是一个评估某 State 好坏的函数，即 $V(s)$ 。比如当前棋盘状态下赢棋的可能性。

初学者容易混淆价值与反馈两个概念，它们都是正负值表达评估的好坏程度，不同点在于 V 评估的是某种 State 的好坏，而 R 评估的是某个 State 下的 Action。

不是所有 Action 都会产生一个 R（当然也可以认为此时的 R 为零），比如象棋中的大多数走子不会吃掉对方某个棋子；但所有的 State 都可以用 V 评估好坏倾向，比如即使在对弈的最初，也可以说此时的 V 略微偏向于先落子的一方。

5. episode 与 step

术语 episode 被用于描述一次 Agent 执行的任务，其由一系列的串联 Action 组成，这些 Action 的执行也被称为 step。对于自动驾驶来说一个 episode 就是一次由起点至终点的完整驾驶，对于围棋来说就是一轮完整的对弈。

通常每个 episode 最终获得的 Reward 被认为是该次 episode 的真实 Value，在训练时该 Value 值通过某种策略被分配给该 episode 中的所有 State 上。从这个角度来说，episode 中间阶段获得的 Reward 是一种短期判定（即编程中的 greedy 策略），而每个 episode 最后获得的 Value 才是需要被优化的终极目标。

10.1.2　基于价值的算法 Q-Learning 与 Sarsa

由于价值（Value）是最终要优化的目标，因此假设已经知道了所有 State 的价值，那么只要在所有行为中选择能产生最大 Value 的那个 Action 就能达到完成最终任务的目的。

强化学习中将这类以寻找到最佳价值函数 $V(s)$ 为目标的算法称为基于价值的算法，典型的有 Q-Learning、Sarsa 等。

1. Q 表

Q-Learning 是最简单、也是最重要的一种基于价值的学习方法，有很多其他算法由它派生而来。Q-Learning 的算法意图是：用一张二维表记录在所有 State 下执行每一个 Action

所能够获得的价值（被称为“Q 值”），并用迭代学习的方法更新其中每个 Q 值。表 10-1 是一个简化了的自动驾驶模型的 Q 表。

表 10-1　Q 表举例

状态	行为		
	向左	正前	向右
路况1	0.5	0.2	0.3
路况2	0.3	0.3	0.6
……	……	……	……

因此，可以将 Q 表看成对函数 $Q(s,a)$ 建模的手段之一，当状态和行为多到无法枚举时，就要选择其他手段表征该函数，暂时还只讨论用 Q 表建模的情况。

2. 基本算法

Q-Learning 的目标就是优化 Q 表中的值，使得 Agent 在每个状态下选择最大 Q 值的行为能够最好地完成任务。而完成这个目标的方法并不奇特，与之前学过的 EM、梯度下降等很多算法类似：先用随机值或零值初始化 Q 表，然后用样本计算结果对 Q 值进行修正，过程如下：

```
1. initialize_Q
2. for each episode:                                    # 每个棋局是一个 episode
   set_initial_state                                    # 设置棋盘初始状态
   for each step:
       s_t = latest state
       a_t = pick an action according to current state   # 走一步棋
       s_{t+1},r = observe the new state and reward  # 观察当前棋盘 state 和 reward
       update Q: Q_new(s_t,a_t) = Q_old(s_t,a_t) + α(r + γ·maxQ(s_{t+1},) − Q_old(s_t,a_t))
```

其中的关键是更新 Q 表的公式：

$$Q_{new}(s_t,a_t)=Q_{old}(s_t,a_t)+\alpha(r+\gamma\cdot\max Q(s_{t+1},)-Q_{old}(s_t,a_t))$$

其中 s_t 是已有状态，a_t 是在 s_t 下选择的 Action，s_{t+1} 是产生的新状态，r 是执行 a_t 所产生的 Reward，$\max Q(s_{t+1},)$ 是当前 Q 表中 s_{t+1} 下所有 Action 中所能获得的最大 Q 值。

3. 学习效率与贴现因子

在上述算法中涉及两个超参数 α 和 γ，它们的取值范围都是 0~1。其中 α 是学习效率，该超参数在之前章节的梯度下降类模型中经常遇到：该值越大 Q 值更新越快，但容易产生

震荡；该值越小则越容易滞留在局部优化区域。

注意：超参数α和γ的命名使用的是希腊字母“alpha”和“gamma”，而非英文“a”和“y”。

第二个超参数是贴现因子γ（discounting factor），它用于定义每个 episode 最终获得的 Value 在多大程度上能传播到之前所有 State 对 Value 的判断。该超参数的产生有一个很自然的理由：很多场景中越靠近确切 Value 产生时的 Action 对该 Value 的产生所做的贡献越大。举个例子：在自动驾驶场景中，一次事故的产生可能只与事故之前几秒钟的 Action 有关，而与几分钟甚至几小时前的驾驶无关。但是也有一些场景并非这样，比如一盘围棋最后的输赢很难说与哪几次落子的关系更大。对于自动驾驶这种 Value 的传播范围更小的情况，应该设置比较小的贴现因子；对于围棋这样每个 Value 可能均匀地受到之前所有 Action 影响的情况，可以设置接近于 1 的贴现因子。

4. 用$\varepsilon-$greedy 加入随机性

在之前的算法中忽略了一个细节：在每个步骤中如何选取合适的 Action，即a_t。这也就是上一节中提到的强化学习中的策略 π。一个很符合直觉的观点是选取在当前 State 下能产生最大 Q 值的 Value 即可，即 max a in all Q(s_t,)。但这会带来如下几个问题。

◎ 导致s_t下除具有最大 Q 值外的其他 Action 没有被选中的机会，实际上使它们变成了死区，也就是在被初始化后永远不会尝试更新，也就达不到算法的优化目的。

◎ 在一些控制系统中，确实有无法精确选择 Action 的情况。比如在自动驾驶中，如果向左转 50°能够获得最大安全，但 49°或 51°就会发生碰撞，那也许 50°也不是一个最佳选择，因为略有误差就会产生很坏的结果。

由于这些问题的存在，在 Agent 选择 Action 时应该加入一些随机性，增强 Action 的鲁棒性。典型的加入随机性的方法是$\varepsilon-greedy$，即定义一个取值范围为 0～1 的超参数ε，使得有$1-\varepsilon$的概率选择产生最大 Q 值的 Action，而另外的ε概率随机选择一个 Action。

5. on-policy 的 Sarsa 算法

现在已经掌握了完整的 Q-Learning 算法，不知读者是否会意识到算法中一个奇怪的现象：在每个步骤中，选择当前行为a_t用的是策略$\varepsilon-greedy$，但在更新 Q 值时实际用的是并不合理的 max 策略。

Q-Learning 这种两种策略不相符的情况被称为 off-policy，相应地也就有了所谓的 on-policy 算法。Sarsa 是在对 Q-Learning 进行 on-policy 改进后所形成的算法，流程如下：

```
1. initialize_Q
2. for each episode:
   set_initial_state
   for each step:
       s_t = latest state
       a_t = pick an action with ε-greedy
       s_{t+1},r = observe the new state and reward
       a_{t+1} = pick an action with ε-greedy     # 用 ε-greedy 估计下一步 action
       update Q: Q_new(s_t,a_t) = Q_old(s_t,a_t) + α(r + γ·Q(s_{t+1},a_{t+1}) - Q_old(s_t,a_t))
```

上述 Sarsa 算法与 Q-Learning 唯一的不同在于流程中的最后两步。Q-Learning 在更新 Q 表时使用 $\max Q(s_{t+1},)$ 估计下一个步骤的 Q 值；而 Saras 则先用 $\varepsilon-greedy$ 估计下一个 step 使用哪一个 action，再用该 Action 的 Q 值作为下一个步骤的 Q 值。

Sarsa 算法的名称来自每个步骤循环内的执行步骤，即 State–Action–Reward–State–Action。在进行了这样的 on-policy 改进后，有利于算法在训练中规避一些危险的优化点（比如“50°能够获得最大安全，但 49°或 51°就会发生碰撞”的例子）。

10.1.3 基于策略的算法

基于价值的算法能够解决简单场景中的很多问题，但有些时候状态类型太多，无法建立一个完整的 Q 表，或者状态是连续值更加无法用离散的表格来描述。对于这样的情况可以尝试使用所谓的基于策略的学习方法。

如前所述，强化学习中的策略是指导 Agent 在每个步骤如何选择 Action 的方法，通常用 Action 的概率分布表达，比如上一节中学习的 $\varepsilon-greedy$ 就是一种简单的策略。基于策略的方法原理是：对于能获得正向结果的 Action，增加它们在今后被选中的概率；反之则降低。

1. 马尔可夫决策过程（MDP）

基于策略的方法用一个 Action 的概率函数进行建模，即函数 $\pi(s)$ 输出一个向量表达在 s 状态下每一种 Action 的可能性，或者用函数 $\pi(s,a)$ 输出一个标量返回在状态 s 下选择行动 a 的可能性。

策略概率函数π的训练将一个 epsoide 看成一个马尔可夫决策过程（Markov decision processes，MDP），每个步骤是一个马尔可夫结点，所有状态之间有一个基于 Action 的转换概率。如图 10-1 所示描述了一个具有两种 Action 和三种状态的简单的 MDP。

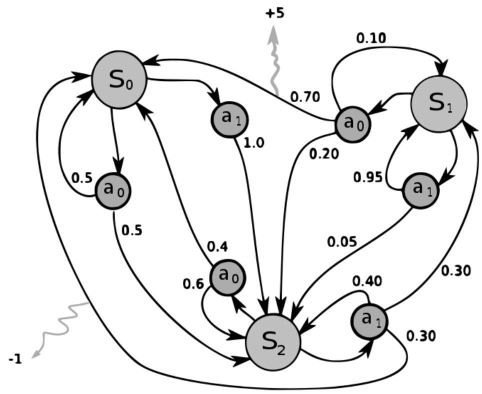


图 10-1　MDP 模型举例（取自 Wiki 百科）

如图 10-1 所示，MDP 的每个状态 s 都有一个概率分布描述可以采取哪些 Action，用 MDP 模型学习大量的 episode 可以逐渐优化得出图中的所有概率值（也就是策略函数π）。

2. Policy Gradient

现在来学习一种解决 MDP 问题的具体方法——Policy Gradient。从名称就可以大概知道，该方法是一种与神经网络训练类似的按样本产生的参数梯度进行参数修正的算法。Policy gradient 假设策略函数π有参数集θ，训练的目的就是调整θ的值使得 Agent 在按照π执行每个 episode 的时候获得尽可能大的 Value。梯度类算法都需要一个损失函数作为优化手段，一种简单的 Policy Gradient 损失函数定义为：

$$\mathcal{J}(\theta)=\sum_s\sum_a \pi(s,a)\cdot V(s,a)$$

其中 V 是价值函数，靠直觉就可以知道，“给产生大 Value 的（State，Action）分配高的π可能性、给小 Value 的（State，Action）分配低的π可能性”可以最大化$\mathcal{J}(\theta)$。因此 Policy Gradien 的目的是最大化损失函数$\mathcal{J}(\theta)$，用蒙特卡洛方法模拟大量的 epsoide，损失函数优化原理如下：

```
1. initialize_θ
2. for each episode:
  set_initial_state
```

```
    for each step:
        pick_action_by_π                              # 根据策略概率分布选择 Action
        θ ← θ + α ∂J(θ)/∂θ                            # 梯度修正
```

其中 α 仍然是学习效率超参数，$\frac{\partial \mathcal{J}(\theta)}{\partial \theta}$ 是损失函数梯度。对于不同的策略概率函数（比如高斯分布、多项式分布、神经网络等）$\frac{\partial \mathcal{J}(\theta)}{\partial \theta}$ 的具体形式会有不同，所以这里不再展开。但从 $\mathcal{J}(\theta)$ 的定义可以知道，Policy Gradient 的优化仍然由价值 V 驱动，但与基于价值的算法不同，它不需要优化一个价值表，而是直接优化策略函数的参数集 θ。

因此，基于策略类算法可以对具有无法枚举的连续 State、Action 场景进行建模，而这是需要维护 Q 表的 Q-Learning、Sarsa 等无法做到的。

3. Actor-Critic

虽然 Policy Gradient 达到了对连续 State 和 Action 建模的目的，但它实际是一种间接方法（在训练中利用价值函数作为隐藏函数），使得 Policy Gradient 的收敛速度很慢。

这促使了一种综合两个策略优势的算法的产生，即 Actor-Critic。Actor-Critic 同时学习每个 episode 中 π 和 Value，用预测的 Value 指导 π 的梯度修正，并用预测的策略值 π 指导 Agent 在每个步骤的行动。

通过这种方式，Actor-Critic 将原本隐藏在幕后的 Value 也拉到了前台，使得 π 和 Value 可以互相监督齐头并进。因此 Actor-Critic 是当前强化学习被应用最成功的算法，本章最后的案例将详细解析基于深度学习的 Actor-Critic 算法实现。

10.1.4 基于模型的算法

传统意义上强化模型的算法被分为三大类：基于价值算法、基于策略算法、基于模型算法。之前已经学习了前两类算法，它们都是用于指导 Agent 如何完成最佳 Action 的算法，但第三类算法却并非用于该目的。

基于模型类的算法的目标并非 Agent 与它的 Actoin，而是 Environment。它的目的是建立一个与真实场景类似的环境模型，使得在收到来自 Agent 的指令后能够给出正确的 State 与 Reward。

这对于电子游戏等 Agent 来说也许是不必要的，因为电子游戏本身就是一个虚拟环境，无须再复制另一个游戏，但对于物理环境中的大多数 Agent 来说这类算法几乎是必需的：强化学习需要演练大量的 episode，但是物理世界中几乎没有免费的可重复环境供 Agent 进行尝试。很难想象一个自动驾驶 Agent 在训练时需要真的跑在马路上横冲直撞，或者 AlphaGo 需要有真实围棋专家进行陪练。因此，三类算法在强化学习训练中的关系如图 10-2 所示。

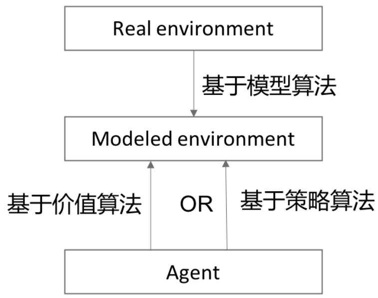


图 10-2　三类算法在强化学习训练中的关系

如图 10-2 所示，大多数的 Agent 只是在虚拟环境中用价值或策略算法进行训练，只有在应用时才会用于真实环境。

对于完全已知的白盒环境，可以用规则定义的方法建立虚拟环境，比如围棋博弈。而对于无法明确定义的黑盒环境，则基于模型类算法变成一个有监督学习的问题，即记录大量真实环境中的 Action（State，Reward）的映射样本，将它们作为特征和标签训练出较好的虚拟环境。

无论如何，基于模型类的算法都与具体的真实场景紧密相连。因此本章不过多关注这类方法，而着重讲解 Agent 在虚拟环境中的强化学习方法。

10.2　OpenAI Gym

对于初学者来说，强化学习令人生畏的一个原因可能是其对环境的要求较高。对于有监督和无监督学习来说，只需要下载一些静态数据即可开始实践各种模型，而强化学习却需要一个能根据 Action 提供 State 和 Reward 的动态虚拟环境。本节介绍一个提供了这样环境的开源组件——OpenAI Gym。

10.2.1 环境调用

OpenAI 是一家著名的人工智能非营利组织，它专注于前沿的机器学习技术，并将其研究成果进行开源分享以便更多人参与进来。OpenAI Gym 是专注于强化学习的工具包，它提供了很多游戏、模拟控制的实验环境，即使是 DeepMind 发表的很多研究也是围绕 OpenAI Gym 的某个环境而进行的。图 10-3 是官网上展示的一组 2D 自动控制虚拟环境。

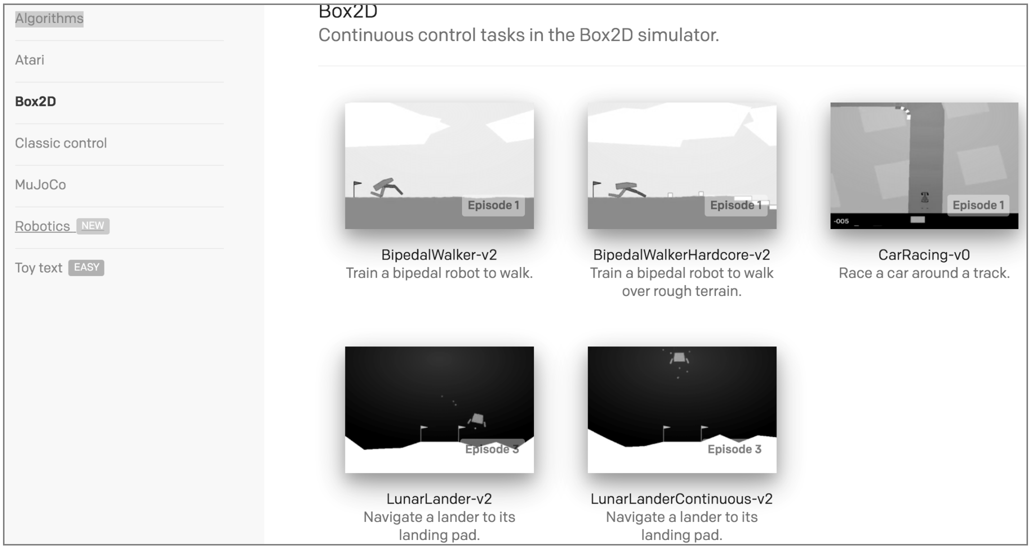


图 10-3　2D 自动控制虚拟环境举例（取自 http://gym.openai.com）

在图 10-3 中列举了 Box2D 中机器人行走、赛车、飞行器着陆等虚拟环境。在左侧的导航栏中，选择 Atari（电子游戏）、Robotics（机械手臂）、Toytext（文本环境）等可以浏览到更丰富的内容。

1. 安装

安装 OpenAI Gym 一般可分为两步：在操作系统中安装必要的图形图像工具，然后用 pip 命令安装 Python 组件。以 MacOS 举例，首先用 brew 命令安装操作系统工具：

```
# brew install cmake boost boost-python sdl2 swig wget
```

然后安装 OpenAI 的 Python 组件库：

```
# export MACOSX_DEPLOYMENT_TARGET=10.12                    # 配置编译环境
# pip install 'gym[all]'                                   # 安装所有 Gym 组件
```

如果安装成功，打开 Python 命令行已经可以引入 Gym 组件：

```
# python3
>>> import gym
>>> gym.__version__
'0.9.2'
```

如上所示，本书使用最新的版本 OpenAI Gym 0.9.2。

注意：如果需要在其他操作系统中安装，或使用 Robotics 组中的环境，需要参考官网中更详细的安装指南。

2. 构造与显示环境

OpenAI Gym 对所有环境提供了几乎一致的使用接口，比如所有环境可以通过 make()函数构造，然后用环境对象的 reset()、render()方法进行重置与显示，比如：

```
>>> env = gym.make('CartPole-v0')          # 传入环境名
>>> env.reset()                            # 通常在每个 episode 开始调用
>>> env.render()                           # 显示
```

上述代码可以显示如图 10-4 所示的 CartPole 环境初始状态。

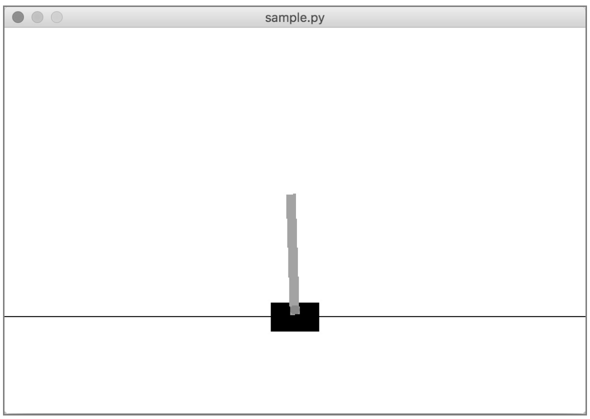


图 10-4　CartPole 环境初始状态

该环境是一个直立在黑色方块上的未固定长杆，Agent 的任务是控制黑色方块的移动，使得长杆不至于倒下。同样，也可以用类似的简单代码显示登陆飞行器着陆环境 LunarLander-v2，代码如下。

```
>>> env = gym.make('LunarLander-v2')
>>> env.reset()
>>> env.render()
```

该环境初始状态如图 10-5 所示。

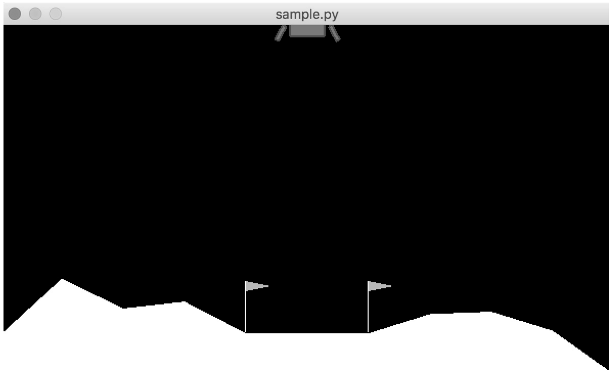


图 10-5 LunarLander-v2 初始状态

当前所有可用的环境名可以在所安装 OpenAI Gym Python 组件的 gym/envs/__init__.py 文件中找到。

3. 执行与观测

建立了静态环境之后，就可以控制 Agent 如何行动并获取反馈了，所有环境都可以通过 step()函数完成该动作，比如：

```
action = env.action_space.sample()                    # 随机选择一个 Action
observation, reward, done, info = env.step(action) # 将 Action 传入 step()执行，并读结果
```

上述代码随机地指定了一个 Action，并将其作为参数传入 step()函数执行，该函数在执行完成后返回如下四个参数。

◎ Observation（object）：环境状态，也就是 10.1.1 节所说的 State。

◎ Reward（float）：Action 执行后所获得的 Reward，可以用负值表示执行代价。

◎ Done（boolean）：是否一个 eposide 已经完成。当 done==True 时 Reward 返回值给出了最终 Value 值。

◎　Info（dict）：调试信息。

结合 reset()和 render()，下述代码可以产生执行一个 eposide 的动画效果：

```
import gym
env = gym.make('CartPole-v0')
env.reset()                                              # 开始一个 eposide
env.render()                                             # 显示初始 environment
done = False
while not done:
    # 随机执行一步
    observation, reward, done, info = env.step(env.action_space.sample())
    env.render()                                         # 显示执行后的 environment
```

由于只是随机的选择动作，该 eposide 很快就以失败告终了。

4. Action 参数与 Observation 参数的取值

在上述代码中，只是用环境中的内置函数 env.action_space.sample()选取随机参数 Action，并没有深究该 Action 的取值范围。在 Gym 中不同环境会有不同的 Action 定义，可以用如下语句查看某环境的 Action 类型：

```
>>> print(env.action_space)
Discrete(2)
```

上述代码得到的结果是一个二值离散类型，即该环境的 Action 只能取值 0 或 1，而每个值的具体含义（比如：向左/向右，向上/向下）由环境在 step()函数中自行解析。

类似的，不同环境的状态变量 observation 也有不同定义，可通过如下语句查询：

```
>>> print(env.observation_space)
Box(4,)
```

该语句获得的是一个四维的 Box 类型。Gym 中的 Box 是一个连续值的坐标系，可通过 high 和 low 属性查看该坐标系的上限与下限：

```
>>> print(env.observation_space.high)
[4.8 3.4 4.2 3.4]
>>> print(env.observation_space.low)
[-4.8 -3.4 -4.2 -3.4]
```

本例结果显示，任何取值范围在[–4.8~4.8，–3.4~3.4，–4.2~42，–3.4~3.4]之间的向量

都是 observation 的合法取值。

Gym 的大多数环境都用 Discrete（离散）和 Box（多维连续）定义了 Actin 与 observation 的类型。偶有遇到其他类型的，可以查看 gym/spaces/*.py 中的源码寻找其具体定义。

10.2.2 实战：用 Q-Learning 开发走迷宫机器人

在 Gym 的 Tony text 环境组里，有一个冰湖游戏 FrozenLake-v0。它的环境是一个冻冰的湖面，其中有很多冰洞，Agent 的任务是从湖的一点出发走到湖面的另外一点。在该过程中如果误入冰洞，则该项任务失败。由于这个任务类似传统的迷宫游戏，所以本节称该 Agent 为"会走迷宫的机器人"。

1. 环境介绍

出于简单考虑，该环境完全用文本表达，构造环境后调用 render()函数，可以在控制台中发现如图 10-6 所示的效果。

```
SFFF
FHFH
FFFH
HFFG
```

图 10-6 FrozenLake-v0 初始状态

图 10-6 中用一个字母矩阵描述了湖面状态，其中共有 16 个 Agent 可以行走的位置。

◎ S 是 Agent 的出发点。

◎ F 是 Agent 可以安全行走的位置。

◎ H 是冰洞，Agent 如果落入冰洞则游戏以失败结束。

◎ G 是 Agent 的终点，Agent 到达该点则游戏胜利。

2. 算法选择

接下来，查看该环境 Action 与 Observation 的变量类型：

```
>>> print(env.action_space)                    # 查看 Action 类型
Discrete(4)
>>> print(env.observation_space)               # 查看 observation 类型
```

```
Discrete(16)
```

可知 Action 与 observation 都是离散变量。不难想象，Action 的 4 个离散值分别代表 Agent 向前、后、左、右 4 个方向行走，而 observation 的 16 个值代表 Agent 的当前位置。

注意：对于 Agent 来说，它只能“看到”自身的位置，但无法知道前后左右哪里是安全点或是冰洞点，因此走出湖面对它来说并不容易。

在上一节中已经分析，基于价值的算法非常适合对离散变量环境的建模，因此这里选择 Q-Learning 算法解决该任务。

3. 实现算法

Q-Learning 是基于 Q 表的迭代更新算法，实现起来并不困难，代码如下：

```
import gym
import numpy as np

env = gym.make('FrozenLake-v0')                        # 构建环境

Q = np.zeros([env.observation_space.n,env.action_space.n]) # 初始化 Q 表
lr = .5                                                # 学习速度α
y = .95                                                # 贴现因子γ

for i_episode in range(200000):                        # episode 循环
    s = env.reset()                                    # 每个 episode 开始时 reset
    for t in range(1000):                              # step 循环
        env.render()
        a = np.argmax(Q[s,:] +
                np.random.randn(1,env.action_space.n)
                * max(1./(i_episode+1), 0.01))     # ε-greedy 策略
        s1, r, d, info = env.step(a)                   # 执行 Action 并读取结果
        Q[s,a] = Q[s,a] + lr*(r + y*np.max(Q[s1,:]) - Q[s,a])   # 更新 Q 表
        s = s1
        if d == True:                          # 如果游戏结束，开始下一个 episode
            break
```

上述代码几乎应用了 Q-Learning 中的所有概念，程序主体是一个两层循环，在每一个 episode 的每一个步骤中执行 Action 并更新 Q 表，具体解析如下：

◎　变量 Q 以 Numpy 数组形式定义了 Q 表，数组形状为[env.observation_space.n，

env.action_space.n]。

◎ 变量 lr、y 定义了超参数学习速度 α 、贴现因子 γ 。

◎ 在每一个步骤中，选取 Action 的策略是用了 ε-greedy，使用的 ε 值为动态的 1./(i_episode+1)，即随着 episode 的增加随进性不断减小。

◎ 更新 Q 表的公式是：

$$Q_{\text{new}}(s_t,a_t)=Q_{\text{old}}(s_t,a_t)+\alpha(r+\gamma\cdot\max Q(s_{t+1},)-Q_{\text{old}}(s_t,a_t))$$ 。

4. 实验效果

为了观察 Q-Learning 的运行效果，先在运行之前在程序中添加一些统计与显示代码：

```
###### 省略若干代码 #######
success = 0                                    # 最终成功走出迷宫的 episode 数量
for i_episode in range(200000):
    s = env.reset()
    for t in range(100):
        print(chr(27) + "[2J")                 # 清屏
        env.render()
        print("episode: ", i_episode)          # 显示总 episode 数量
        print("success: ", success)            # 显示成功数量
        print("rate: {:.2f}%".format(
          success/max(1, i_episode)*100,))     # 成功率
        a = np.argmax(Q[s,:] +
              np.random.randn(1,env.action_space.n) *(1./(i_episode+1)))
```

现在运行该程序，就可以观测到在训练中 Q-Learning 实验过程，如图 10-7 所示。



图 10-7 Q-Learning 实验过程

随着 episode 的增加，Agent 能够成功找到迷宫出口的几率逐渐增加。在达到约 67% 后逐渐平稳，读者可以尝试修改代码中的超参数，看是否能够训练出更好的成功率。

10.3　深度强化学习

深度强化学习（Deep Reinforcement Learning）是 DeepMind 公司用深度学习工具实现强化学习的一系列算法模型，由于在 AlphaGo 上的成功取得了广泛关注。因此目前深度强化学习的资料来源主要是 DeepMind 首席研究员 David Silver 的论文与演讲稿，本节解读这些资料，为后续实践打下基础。

10.3.1　DQN 及改进

强化学习中有几个重要的环节其实是对未知函数的拟合，比如 Q 表可以看成对价值函数 $V(s)$ 的拟合，Policy Gradient 可以看成对 $\pi(s)$ 的拟合。但当环境中的 State 与 Action 的取值多到无法枚举时，就会感到仅用 Q-Learning 等传统方法的力不从心。

1. 普通 DQN

有监督学习在本质上可以看成对复杂未知函数的拟合，以神经网络为基础的深度学习是近十年来最被广泛应用的有监督机器学习模型。因此，如果用深度神经网络取代 Q 表就可以解决上述问题，这就是所谓的 Deep Q-Networks。如图 10-8 所示是 David Silver 描述的应用在 Atari 电子游戏上的 DQN 中的价值卷积网络。

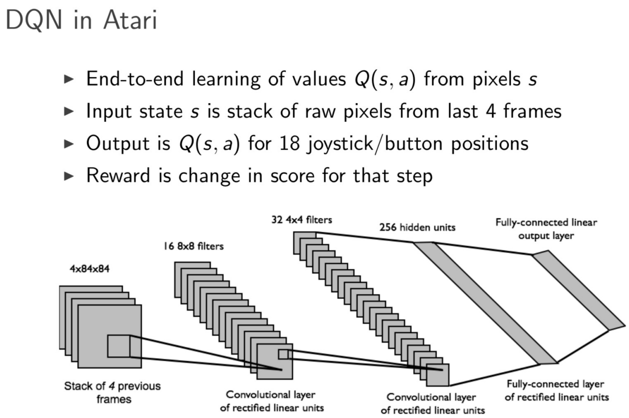


图 10-8　DQN 中的价值卷积网络（取自 David Silver 演讲稿）

该网络几乎是第 9 章中介绍的最经典卷积网络 LeNet 的翻版，都由两个卷积层后接全

连接层构成，只是在 feature map 等的数量上有所增加。这样，将传统 Q-Learning 中在每个步骤中更新 Q 表的步骤替换成对该卷积网络的训练，就形成了 DQN 的强化学习过程。

2. DQN 的改进

普通 DQN 首次于 2013 年 12 月发表在论文 *Playing Atari with Deep Reinforcement Learning* 中，在之后的几年里又出现了几个改进版本，其中最重要的有三个。

- ◎ Double DQN：2015 年 9 月发表于论文 *Deep Reinforcement Learning with Double Q-Learning*，目的在于解决普通 DQN 中的 overestimation 问题，即训练中过高估计后续 Q 值（在计算后续状态 Q 值时使用了 Max 算子）。解决方案是使用两个同构的神经网络，一个具有最新参数的网络用于评价可能的后续状态 Q 值，另一个用于选择当前的下一步。
- ◎ Prioritized Replay DQN：2015 年 11 月发表于 *Prioritized Experience Replay*，对训练样本设置了优先级，那些没有被很好预测正确的样本被增加了权重，使得这些样本有更多机会被改善。该思想与第 3 章中学习的 Adaboost 类集成学习方法有异曲同工之处。
- ◎ Dueling Network：2015 年 11 月发表于 *Dueling Network Architectures for Deep Reinforcement Learning*，在 DQN 网络的最后一层将预测拆分为两部分，一个是对当前 State 的 Q 值预测，另一个对在当前 State 采取指定 Action 后对 Q 值增幅/减幅的预测。该思想类似深度学习中的 ResNet，其效果也是能够使网络更快收敛。

经过这些改进后，价值类深度学习算法渐臻成熟。在 DeepMind 的论文中还列举了一些 DQN 的实验数据，有兴趣的读者可参考 David Silver 的教学网站。

10.3.2 DPN、DDPG 及 A3C

与 DQN 是深度学习的基于价值类算法一样，DeepMind 也提出了基于策略的深度学习算法——从 DPN、DDPG 到 A3C。

1. DPN（Deep Policy Network）

DPN 可以简单地看成当使用神经网络对 $\pi(s)$ 函数建模时的 Policy Gradient 算法。David Silver 给出的算法如图 10-9 所示。

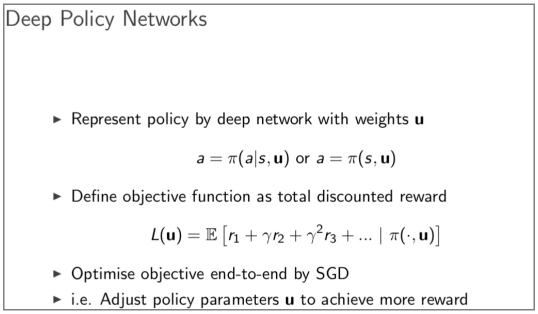


图 10-9　Deep Policy Network（取自 David Silver 演讲稿）

其中的神经网络权重参数 u 就是在 10.1.3 节中提到的策略函数参数集 θ； $L(u)$ 就是 10.1.3 节中提到的损失函数 $\mathcal{J}(\theta)$。

这里 $L(u)$ 的定义使用了带贴现因子的价值期望形式，即 $E[r_1 + \gamma r_2 + \gamma^2 r_3 + \ldots | \pi]$。其中 γ 是与 Q-Learning 算法中含义相同的贴现因子， r_1 是当前状态获得的 Reward， r_2 是下一个状态获得的 Reward， r_3 是再下一个状态获得的 Reward……其含义也就是，越是远期的 Reward，对当前 Value 的贡献越小。

注意：由于 $L(u)$ 的计算用到了从当前状态开始到 episode 结束的所有 Reward，所以 DPN（包括其他 Policy Gradient 算法）都只能在整个 episode 结束后才能更新策略参数。这也就是所谓的“回合更新制”；相对应的价值类算法则可以实时更新，也就是“单步更新制”。

2. DDPG

DDPG（Deep Deterministic Policy Gradient）是 David Silver 等人 2014 年发布的论文，它实现的是深度学习版的 Actor-Critic 算法，其原型如图 10-10 所示。

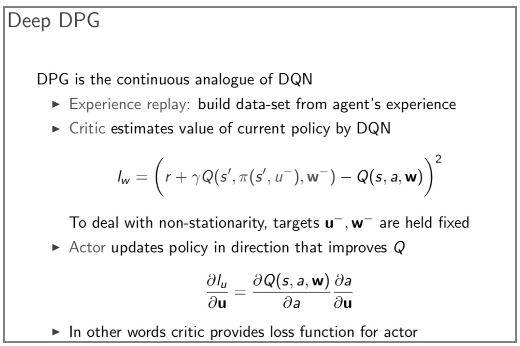


图 10-10　Deep DPG（取自 David Silver 演讲稿）

图 10-10 中名词解释如下。

◎ Experience replay：将历史的 State、Action、Reward 样本保存起来，供深度网络在训练中反复使用。这一方面降低了重新采样的成本，更重要的是能用样本 shuffle 等方法解决同一 episode 样本连续训练产生的泛化不足问题。

◎ Critic：即价值学习部分，用深度网络学习 Q 价值函数。

◎ Actor：即策略学习部分，用深度网络学习策略函数。

再次强调，在 Critic-Actor 中用预测的 Value 指导 π 的梯度修正，并用预测的策略值 π 指导 Agent 在每个阶段的行动。

3. A3C

A3C（Asynchronous Advantage Actor-Critic）来自 DeepMind 团队于 2016 年 6 月发表的论文 *Asynchronous Methods for Deep Reinforcement Learning*。像其简写名称 A3C（三个 A 和一个 C）暗示的那样，可以通过解析三个 A 的含义理解整个模型。

◎ Asynchronous：异步训练，用多个 Agent 在各自环境中独立训练，然后将所有训练时的梯度优化同时应用在一个综合模型中。

◎ Actor-Critic：也就是 DDPG 中用深度学习实现的价值与策略结合的算法。

◎ Advantage：在价值学习方面，使用差分学习策略 $A(s,a)=Q(s,a)-V(s)$，即学习 Action 本身导致的价值增幅/减幅，该思想与 DQN 中的 Dueling Network 方法类似。

总结上述内容如图 10-11 所示。

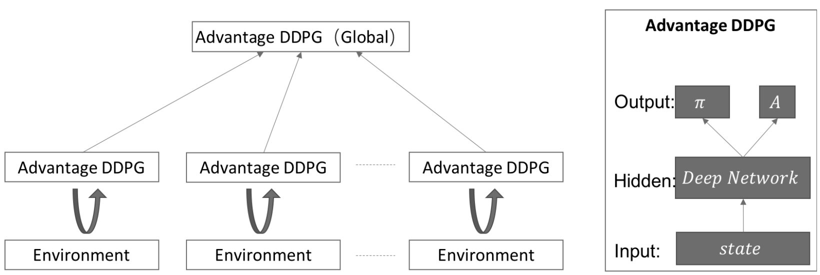


图 10-11　A3C 模型架构

总的来说，A3C 是对之前几种深度强化学习模型的综合。在使用了异步架构后增强了样本的多样性，同时使得分布式计算训练效率的提高成为可能。

10.3.3　实战：用 DPN 训练月球定点登陆

设想有一个月球定点登陆计划，其任务要求是在月球上空释放登陆器，让其在着陆过程中自行调节若干飞行引擎，使得自己可以降落在指定地点。任务的难点在于，飞行器的初始速度与角度不受控制，并且需要适应月球崎岖不平的地面环境。如果偏离指定地点或速度太快导致登陆器损坏，都会导致任务失败。

本案例用 DPN 模型训练一个 Agent 解决上述问题，模型代码源自 Juliani 的 GitHub 开源程序，这里略作修改使其能适应更多的 OpenAI Gym 环境，并通过飞行器登录环境“LunarLander-v2”解析 DPN 的具体编程方法及运行效果。

说明：代码源 Git 地址：https://github.com/awjuliani/DeepRL-Agents/blob/master/Vanilla-Policy.ipynb。

1. 环境介绍

首先通过环境的 observation_space 和 action_space 属性查看 State 和 Action 的变量类型：

```
>>> import gym
>>> env = gym.make('LunarLander-v2')
>>> print(env.action_space)
Discrete(4)
>>> print(env.observation_space)
Box(8,)
```

可知该环境的 Action 是一个四值离散整数，而 State 是一个八维连续值数组。至此已经可以开始强化学习建模，但大多读者可能还是会好奇，这些值在环境中的物理意义是什么？

OpenAI Gym 的所有环境没有太多的文档说明，但由于其代码结构简单，通常可通过阅读源代码了解这方面的信息。'LunarLander-v2'的代码路径为 gym/envs/box2d/lunar_lander.py，观察 step()函数可以知道，Action 的四个值中，0 代表不开动任何引擎，1~3 代表开动三个引擎之一，而 State 的含义如图 10-12 所示。

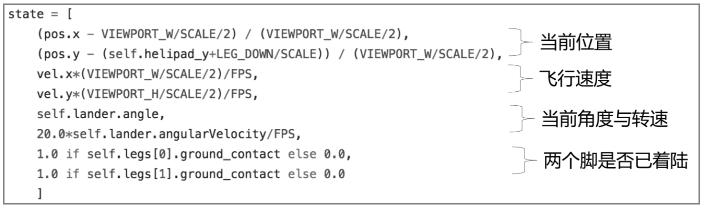


图 10-12 'LunarLander-v2'中 State 的物理含义

2. 构建神经网络

理解了 State 与 Action 的类型与含义后，就可以着手搭建 DPN 中的神经网络了。该网络的输入层就是环境中的 State，输出层则是预测的 Action 策略概率分布，同时样本中的 Reward 和 Action 用于计算网络的损失值，代码如下：

```
class agent():
def __init__(self, lr, s_size, a_size, h_size):
    ################ 下面是网络静态结构定义 ####################
        self.state_in = tf.placeholder(shape=[None, s_size], # 输入层占位器
            dtype=tf.float32)
        hidden = slim.fully_connected(                          # 隐藏层
            self.state_in,
            h_size,
            biases_initializer=None,
            activation_fn=tf.nn.relu)
        self.output = slim.fully_connected(                     # 输出层
            hidden,
            a_size,
            activation_fn=tf.nn.softmax,
            biases_initializer=None)
        self.chosen_action = tf.argmax(self.output, 1)          # 最优选 Action

        ################# 下面是损失定义与训练优化 #####################
        # Reward 占位器
        self.reward_holder = tf.placeholder(shape=[None], dtype=tf.float32)
        # Action 占位器
        self.action_holder = tf.placeholder(shape=[None], dtype=tf.int32)

        # Action 的索引
        self.indexes = tf.range(0,tf.shape(self.output)[0]) *
                    tf.shape(self.output)[1] + self.action_holder
```

```
        # 输出层中 Action 的值
        self.responsible_outputs = tf.gather(
            tf.reshape(self.output, [-1]), self.indexes)
        # 损失 ←−log(π(s,a))×R
        self.loss = -.reduce_mean(
            tf.log(self.responsible_outputs)*self.reward_holder)

        tvars = tf.trainable_variables()
        self.gradient_holders = []
        for idx, var in enumerate(tvars):
            placeholder = tf.placeholder(tf.float32,
                  name=str(idx) + '_holder')
            self.gradient_holders.append(placeholder)

        self.gradients = tf.gradients(self.loss, tvars)

        optimizer = tf.train.AdamOptimizer(learning_rate=lr) # Adam 优化器
        self.update_batch = optimizer.apply_gradients(
            zip(self.gradient_holders, tvars))
```

上述代码整体上看分为两部分，第一部分定义网络的静态层次结构，第二部分为损失定义与优化器配置。值得注意的是损失值计算方式的定义，该代码实际上定义了 $J=-\log(\pi(s,a))\times R(s,a)$ 的损失值，由于优化器的目标是最小化该值，因此实际效果是最大化了 $\log(\pi(s,a))\times R(s,a)$，即“给大的 R 分配大的 π 、给小的 R 分配小的 π ”。

3. 回合更新

这里进入 DPN 的主功能，遍历 episode 并进行策略优化，代码如下：

```
myAgent = agent(
    lr=1e-2,                                   # learning rate
    s_size=env.observation_space.shape[0],    # State 的维度
    a_size=env.action_space.n,                 # Action 的维度
h_size=8)                                      # 隐藏层维度，可以是任意值

with tf.Session() as sess:
    sess.run(init)                             # TensorFlow 初始化
    i = 0
    gradBuffer = sess.run(tf.trainable_variables())
    for ix, grad in enumerate(gradBuffer):
        gradBuffer[ix] = grad * 0
```

```
    while i < total_episodes:               # 遍历 episode
        s = env.reset()
        running_reward = 0
        ep_history = []
        for j in range(max_ep):
            a_dist = sess.run(              # 获取当前 State 的 π 分布
                myAgent.output, feed_dict={myAgent.state_in: [s]})
            a = np.random.choice(a_dist[0], p=a_dist[0])
            a = np.argmax(a_dist == a)      # 具有最大概率的 Action

            s1, r, d, _ = env.step(a)       # 执行 step
            env.render()
            # 保存执行历史，准备 episode 结束时训练使用
            ep_history.append([s, a, r, s1])
            s = s1
            running_reward += r
            if d:                           # 如果 episode 结束……
                ep_history = np.array(ep_history)
                # 带贴现的 Reward 计算
                ep_history[:, 2] = discount_rewards(ep_history[:, 2])
                feed_dict = {
                    myAgent.reward_holder: ep_history[:, 2],
                    myAgent.action_holder: ep_history[:, 1],
                    myAgent.state_in: np.vstack(ep_history[:, 0])
                }
                # 训练网络
                grads = sess.run(myAgent.gradients, feed_dict=feed_dict)
                for idx, grad in enumerate(grads):
                    gradBuffer[idx] += grad

                if i % update_frequency == 0 and i != 0:
                    feed_dict = dictionary = dict(
                        zip(myAgent.gradient_holders, gradBuffer))
                    _ = sess.run(myAgent.update_batch, feed_dict=feed_dict)
                    for ix, grad in enumerate(gradBuffer):
                        gradBuffer[ix] = grad * 0
                break
```

程序首先构造了一个 Agent 对象，即上节定义的神经网络模型，然后开启双层循环在环境中采样并训练 DPN 网络。由于基于策略类型的算法只能在 episode 结束后进行回合制

的参数集更新，所以需要在每个 step 保存样本，然后在 d==True 时一起提取出并训练。

值得关注的是在设置训练集 feed_dict 的 reward_holder 之前使用了 discount_rewards() 函数计算带贴现的 Reward 值，代码如下：

```
gamma = 0.99                                          # 贴现因子

def discount_rewards(r):
    discounted_r = np.zeros_like(r)
    running_add = 0
    # 逐个向前，将当前 Reward 传播给之前的所有 step，累加入它们的 Value。
    for t in reversed(xrange(0, r.size)):
        running_add = running_add * gamma + r[t]
        discounted_r[t] = running_add
    return discounted_r
```

上述代码实现了图 10-10 的对 David Silver 演讲稿中 $L(u)$ 的计算。

4. 效果演示

本节完整代码保存在文件 sample_dpn.py 中，可以直接运行该文件观察通过训练 Agent 着陆达到的改善情况。执行如下命令开始月球登陆学习：

```
# python3 sample_dpn.py
```

如图 10-13 所示是程序执行过程中的一些截图，图中两个小旗之间为目标着陆位置。如图 10-13 所示，在训练开始时登陆器常常很快飞出屏幕或跌落到错误地点；随着训练的进行，登陆器慢慢学会控制自己的速度与角度，着陆准确度逐渐提高。

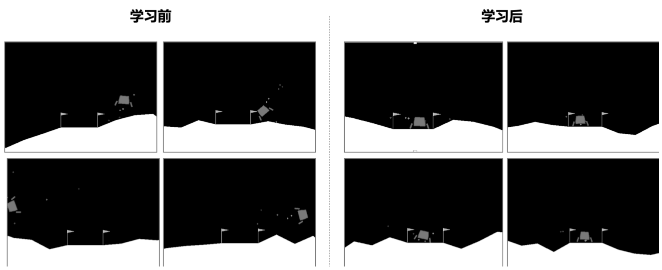


图 10-13　DPN 在“LunarLander-v2”上的学习效果图

读者在实验中可尝试调节程序中的超参数学习效率、贴现因子，甚至是损失函数的计算方式，看是否能达到更好的学习效果。

10.4 博弈原理

博弈原理属于搜索算法范畴，是传统人工智能中主要的解决方案。本节介绍这方面的理论基础，使读者能够理解 AlphaGo 使用的重要博弈武器——蒙特卡洛搜索树，为下一节的博弈机器人实践打下基础。

10.4.1 深度搜索与广度搜索

计算机专业出身的软件开发者可能对数据结构中的深度与广度搜索并不陌生，它们是对树状或图状数据结构进行特定内容搜索的两种策略。然而并不被熟知的是，它们也是解决人工智能问题的基本方法。

如何设计旅途的最佳线路、如何最快走出迷宫、各种棋类博弈，这些被普遍认为是人工智能的典型应用，它们是如何与“搜索”挂上钩的呢？如图 10-14 所示演示了如何将旅行路线问题转化为树的搜索问题。

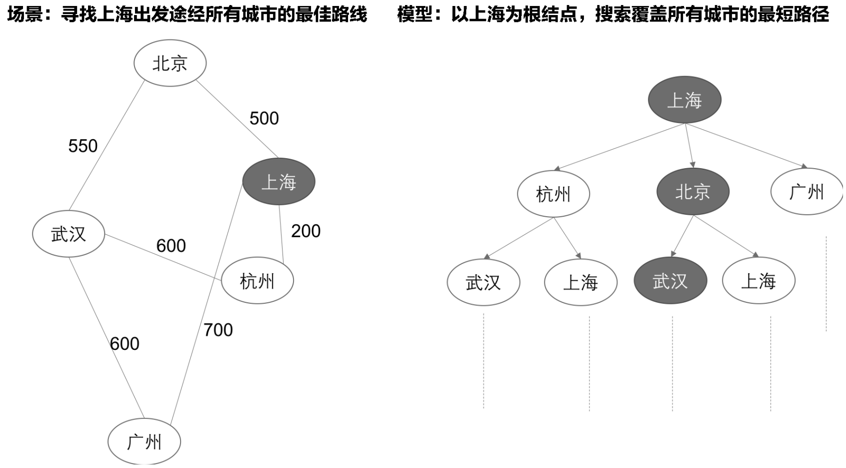


图 10-14　如何将旅行路线问题转化为树的搜索问题

在图 10-14 中，左侧是几个城市的交通费用权重图，Agent 的目的是寻找花费较少（比如总费用小于 2600）的旅行线路；右图是 Agent 解决该问题所用的树形结构。在该场景中问题目标变成了一个搜索问题——遍历所有从根结点到叶子结点的路径，找到其中符合要

求的一条路径。

大多数的人工智能应用场景都可以用类似的方式建模：定义 Agent（代理）的 Action（行动）、State（状态）、Goal（目标），然后构建出一副“以 Action 为边、以 State 为点”的树结构，让 Agent 在该树中搜索符合 Goal 定义的路径。完成了这样的场景建模后，接下来就是算法层面的问题了。

◎ 深度搜索：从树的根结点开始逐个结点向下搜索，如遇叶子结点（或所有子结点已经遍历过）则返回上层结点。在该过程中每进入一个新结点就检测是否已经完成目标，在编程中通常可用“堆栈”实现该算法。

◎ 广度搜索：从根结点开始遍历每一个子结点，在该层所有子结点均已遍历完成后才进入下一层子结点。同样，在该过程中每进入一个新结点就检测是否已经完成目标，在编程中通常可用“队列”实现该算法。

两种策略均可完成对特定 Goal 的搜索，但侧重点有所不同。

◎ 深度搜索使用的是一种贪婪策略：每一步都希望向更深处探索，因此只能选择一个局部看上去最好的子结点。

◎ 广度搜索是一种更稳健的策略：每一步都遍历当前层的所有结点，然后才继续向下一层探索。

从结果的角度看，广度搜索似乎是更好的策略，因为它能保证找到层次最浅的解。在 Agent 的每次 Action 都需要消耗某种代价的情况下，浅层次的解一般是更优选的解。

那么是否深度搜索就没有用武之地了呢？

答案恰好相反，在人工智能中深度搜索反而被应用得更多。这是为什么呢？

“失之东隅，收之桑榆”，在计算机系统普遍存在的效果与效率两个约束中，深度搜索占据了效率一端。实际中的人工智能问题往往不会像图 10-14 那样仅有个位数的 Action 与 State，这意味着问题树是宽而深的。广度搜索既在局部看起来就很差的选择上浪费了很多时间，又要求有大量的空间保存当前层的已计算结果，这导致很少有系统能有足够的资源支撑广度搜索。

这样，在有限的时间和空间上寻找到一个可接受的（虽然可能不是最好的）解成为普遍存在的合理选择。

10.4.2 完美决策

博弈（game）一词来源于棋类，是指具有理性思维的双方选择合规行为以达到击败对方目的的过程。在博弈中的两个 Agent 得到的状态不再仅仅依赖于自身的行为，还需要考虑对方，因此比普通的搜索问题更具有挑战性。

1. 博弈表达

虽然如此，博弈问题归根结底仍然是一棵搜索树，只是在这棵树中选择行动的主体变成了两个。它们带着完全相反的目标轮流行动，先达到目的的一方为胜者，这可表达为一棵博弈搜索树，如图 10-15 所示。

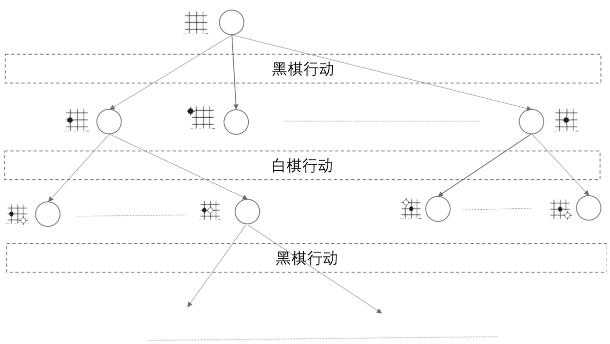


图 10-15 博弈对抗搜索树

在图 10-15 中的根结点是棋盘的初始状态，此时由先行方（围棋中为黑棋）选择行动，因此树的第二层是黑棋第一步落子后可能的状态。接下来轮到白棋行动，即第三层是所有白棋可能的落子。如此循环，形成了一个夹心饼干状的对抗搜索树。

在围棋中黑方的目标是用黑棋占据棋盘中的所有位置，白棋的目标则正好相反。可以量化黑棋的目标为 1，则白棋的目标为−1，最终结果距离哪一方的距离更近则哪一方胜出。在每一步棋中，理性博弈者需要选择最可能达到自己目标的一个棋子付诸行动。

2. 价值计算

根据之前的定义，所有对弈的终结状态（也就是对抗树的叶子结点）都有一个价值评分。那么如何确定非叶子结点的评分呢？

由于理性博弈者都会选择最符合自己利益的行动，因此非叶子结点的价值取决于当前

的行动者是哪一方。

◎　如果是黑方行动：黑方会选择价值最高的一个子结点付诸行动，因此当前结点的价值等于所有子结点中最高的那一个。

◎　如果是白方行动：白方会选择价值最低的一个子结点付诸行动，因此当前结点的价值等于所有子结点中最低的那一个。

如图 10-16 所示是一棵通过这种方式逐级向上确定每个结点价值的对抗树。

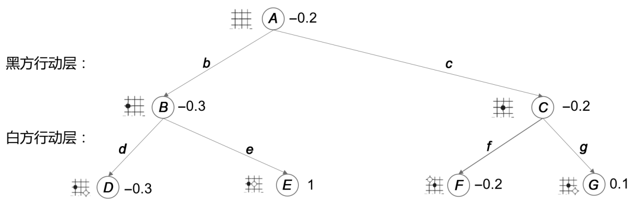


图 10-16　对抗树的价值评估

在图 10-16 中假设已经知道叶子结点 D、E、F、G 的价值，由于它们所在的层次由白方选择行动，因此结点 B 的价值等于价值最低子结点的价值，即–0.3；同理可以得到结点 C 的价值为–0.2。此时再考虑更上一级的结点 A 的价值，由于此时为黑方行动，所以结点 A 的价值等于价值最高子结点，即–0.2。

3. Minmax 算法

通过上面的分析，可以把对抗树中的价值计算看成一个自底向上的过程：由已知价值的叶子结点开始，逐级向上获得所有结点的价值。在理想情况下，对弈者只要遵循价值计算的规律（黑方选择获得最大价值的行动、白方选择获得最小价值的行动）选择适合自己的行动，就可以得到最理想的结果。其实这个方法有一个不错的名字：Minmax 算法（最小最大算法）。

完整的最小最大算法需要遍历对抗树中的所有结点，然后才能获得当前根结点的价值。整个算法的时间复杂度为 $O(b^m)$，其中 b 是每个结点的平均分支数，m 是树的深度。

4. Alpha-Beta 剪枝

Alpha-Beta 剪枝是对 Minmax 算法在性能上的一种改进，因为有些时候无须遍历所有子孙结点即可得到当前结点确切的值。考虑如图 10-17 所示的情况。

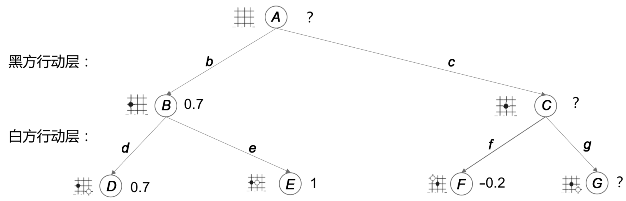


图 10-17　Alpha-Beta 剪枝

当黑方来到图 10-17 中的 *A* 结点时需要评估两个子结点 *B*、*C* 的价值然后选择行动。在经典 Minmax 算法中，此时必须遍历 *B*、*C* 的所有子孙结点。然而其实图 10-17 并不需要这样做：

◎ *B* 点为白方行动结点，因此取 *D*、*E* 中的 min 值 0.7 作为自身价值。

◎ 当遍历到 *F* 点时，发现 *F* 价值为–0.2。由于 *C* 点也是白方行动点，因此可推断 *C* 点本身的价值在[–∞～–0.2]之间。（因为如果 *G* 点的价值大于–0.2，白方不会选择该行动）。

◎ 由于此时 Value(*B*)=0.7、Value(*C*)< –0.2、*A* 点为黑方行动点，所以已经可以断定黑方会选择 *B* 作为其行动点，即 Value(*A*)=0.7。

在上述推理过程中省略了结点 *G* 的价值计算（即将其剪枝），但是取得了与经典 Minmax 同样完美的决策效果，这种改进被称为 Alpha-Beta 剪枝。

10.4.3　蒙特卡洛搜索树

在理论上 Minmax 类算法是能达到最好效果的机器博弈方法，它的算法时间复杂度为 $O(b^m)$。但在实际中，人工智能问题往往有很深（*m* 很大）和很宽（*b* 很大）的树结构，根本没有计算机能够承受的这种指数级别计算资源增长要求。即使是改进了的 Alpha-Beta 剪枝等算法，也只能在线性层面对该计算量打折，无法改变指数级复杂度的事实。

这使得人们不得不寻找一些折中方案（牺牲一些决策准确性，但无须扩展搜索树中的每一个可能结点），蒙特卡洛搜索树（Monte-Carlo tree search，MCTS）是目前被应用最多的这类方法。

在本书第 7 章中已经学习过蒙特卡洛方法，在求解复杂函数时，该方法通过模拟大量样本，对采样结果进行参数估计以达到拟合复杂函数的目的。

而 MCTS 就是将这种用大量随机采样的方法在对抗博弈树中估算结点价值的方法。

MCTS 中的每次采样由如下四个步骤组成。

- ◎ Selection：从当前根结点开始逐步执行贪婪的深度搜索，直到遇到一个尚未扩展的叶子结点为止。
- ◎ Expansion：将该叶子结点扩展，即用所有后续行动生成一组新的叶子结点。
- ◎ Simulation：在新的叶子结点中随机选择一个作为采样行动，估算出该结点的价值。
- ◎ Backpropagation：将行动的结果回传给所有祖先结点，即将最新估算出的价值反应到所有祖先结点上，以影响后续采样的结点选择。

在执行了大量上述采样后，MCTS 选择当前结点的所有子结点中被访问次数最多的一个作为被选择的行动。

通过这种方式在对抗树中选择行动的意义是什么呢？从直觉角度做如下分析：

- ◎ 对抗树最初只有一个结点，而每次采样都能扩展一个状态产生更多结点，因此采样本身是一个使对抗树逐渐生长的过程。
- ◎ 在 Selection 阶段根据子结点的价值执行深度搜索，使得每次采样执行较快。
- ◎ Simulation 阶段无须计算确切的被选择点价值，只需用简单策略并加入随机值进行估算。但由于采样数量很大，根据大数定理采样总体结果会倾向于实现对该点价值进行无偏估计。
- ◎ Backpropagation 阶段使得新的价值估计能影响到所有祖先结点。如果该值较大，则增加了下次采样中这些祖先结点被 Selection 阶段选中的概率；反之则会削弱，使得其他兄弟结点有更多可能被选中。通过这种方式，MCTS 中加入了一定程度的宽度搜索策略。

深度搜索使决策效率更快，宽度搜索使决策更准确。因此可以这样说：MCTS 是结合了深度搜索和宽度搜索、用大数定理寻找最优决策的一种博弈算法。

如果读者现在还对上述内容心存疑惑，不用担心，在下一节案例中有对蒙特卡洛搜索树代码级别的解析。

10.5　实战：中国象棋版 AlphaGo Zero

DeepMind 的 AlphaGo 系列围棋博弈机器人是最有关注度的强化学习应用，在某种程

度上 DeepMind 的技术也代表着当前强化学习领域的最高水平。2017 年 AlphaGo Zero 通过自我学习达到了比初始版本更强的水平，本节解析 AlphaGo Zero 的设计框架与代码，并将改造适用于中国象棋的自我博弈。

需要先说明的是，AlphaGo Zero 击败人类顶尖选手需要数百万盘的自我对抗，显然普通个人无法拥有如此多的计算资源。本节的目的是通过该项目启发读者理解这种结合了强化学习、深度学习、经典博弈的综合应用，然后作为兴趣看看一千盘的自我博弈后能达到怎样的象棋水平。

10.5.1 开源版本 AlphaGo Zero

由于 AlphaGo 获得如此高的关注度，并且 DeepMind 通过论文方式公开了历代 AlphaGo 的技术原理，因此从 2016 年开始有很多组织或个人尝试实现 AlphaGo。

在这些开源实现中，Minigo 是关注度比较高（目前在 GitHub 上有超过 2000 人次的关注量）的一个。它是由谷歌团队基于 DeepMind 论文 *Mastering the game of Go without human knowledge* 用 TensorFlow 实现的 AlphaGo Zero。中国象棋版 AlphaGo Zero 也是基于该论文和 Minigo 项目改造而成的，有兴趣的读者可以先从 GitHub 上尝试下载并使用 Minigo：https://github.com/tensorflow/minigo。

注意：下载和运行 Minigo 并非阅读和实践中国象棋 AlphaGo Zero 的必要步骤，但可以帮助读者快速掌握自我博弈的一些术语和概念。

1. Minigo 项目术语

在 Minigo 的主页 Readme 中主要介绍了该项目的两种体验方式，一是下载已经训练好的模型、观察 AlphaGo 的自我博弈/人机博弈；二是从零开始自己训练模型。这里介绍该项目的一些关键术语，帮助读者能更快上手该项目。

◎ Google Cloud Storage：是一个云文件系统，Minigo 用其保存训练模型和日志文件。如若需要体验有战力的围棋 AlphaGo，需要从其中下载最新模型。

◎ Selfplay：即两个 AlphaGo 之间的自我博弈，它是 AlphaGo Zero 训练过程中产生样本的必要过程，同时也为有兴趣的使用者提供了观察两个 AlphaGo 对局的方式。

◎ GTP Protocol：是一套用文本形式表达和交互围棋对弈过程的协议。虽然文本方

式没有 GUI 图形方式友好易用，但作为研究项目它使得 Minigo 减少了对运行环境的依赖，可以完全通过 Console 终端体验该项目。

◎　Training：即有监督的盘面训练，训练的样本是通过 Selfplay 产生的对弈数据，输出的是神经网络模型。

◎　Validation：在默认的 Selfplay 执行中，Minigo 以 95%的几率将对局保存为训练集，另外 5%的对局保存为验证集。Validation 即是用这 5%的对局验证已训练模型的过程。

2. AlphaGo Zero 架构

在 DeepMind 论文中较清晰地给出了 AphaGo Zero 的整体架构，如图 10-18 所示。

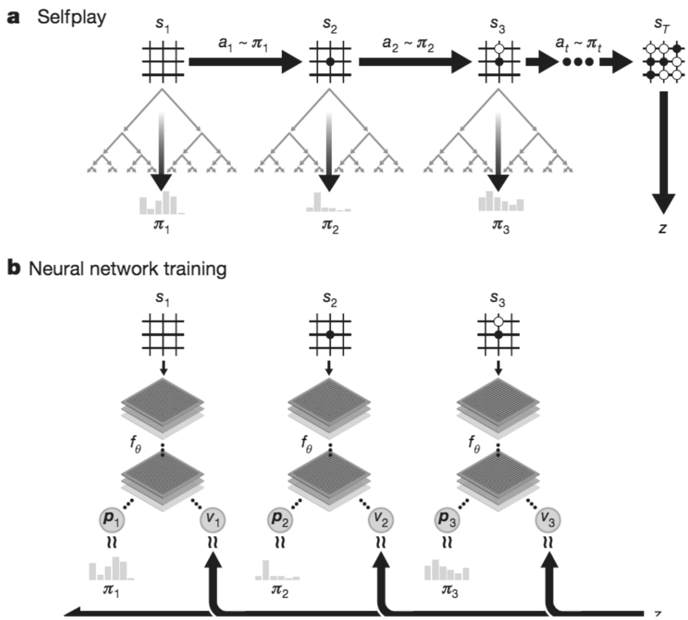


图 10-18　AlphaGo Zero 的整体架构（取自 DeepMind 论文）

图 10-18 中的架构非常简单，由两部分组成：

◎　在 Selfplay 过程中，用蒙特卡洛搜索树（MCTS）生成对局样本。

◎　用神经网络实现深度强化学习。

图 10-18 中的神经网络部分输出层由 P（Policy）和 V（Value）两部分组成，使用了策略与价值结合的深度强化学习算法，即 Deep Deterministic Policy Gradient（DDPG）。

3. 中国象棋的改造思路

从图 10-18 的架构观察，该系统的实现主要基于两大技术：MCTS 和 DDPG，而与围棋本身关联不大。将其改造为中国象棋版本主要涉及以下几个方面的调整。

◎ 为中国象棋盘面的表示建模，从棋子的种类和行棋规则看，这部分工作相比围棋略微复杂。

◎ 用象棋盘面状态改造 MCTS、DDPG 的输入和输出。

◎ 改造 GTP Protocol，用文本化的方式显示盘面状态，并提供人机对弈接口。

◎ 一些细节更改：比如完赛检查、认输判定、样本扩展等。

中国象棋版 AlphaGo Zero（本章后续内容中，Minigo 用于指代中国象棋版 AlphaGo Zero）的完整代码保存在本书所附的源码中，本节后续内容将解析其中的关键代码段，并给出实践指导。

10.5.2 盘面建模

在原始 Minigo 中，只用一个 numpy 二维数组对棋盘建模，用数组元素值–1、0、1 表示每个子位状态：白棋、空、黑棋。而象棋的棋子类型更多，且每一类棋子有不同的行棋规则，因此需设计一个单独模块表达象棋盘面。

关于中国象棋的盘面规则在 GitHub 上也有很多开源实现，这里引用一个结构较清晰的项目：https://github.com/shaochuan/Xiangqi，该项目实现了一个基于 Python 2.7 的在 MacOS 上运行的中国象棋运行环境（没有人机对弈功能）。本书将其改造为 Python 3 版本，并剥离依赖于操作系统的图形界面操作，使其成为独立的象棋规则模块。

1. 棋子

首先需要定义象棋中的若干棋子类型，并用棋盘描述对弈的中间状态，这部分代码保存在 Xiangqi/widget.py 文件中。棋子代码如下：

```
# 棋子中文名称
```

```
red_labels = ["帅", "仕", "相", "傌", "俥", "炮", "兵"]
black_labels = ["將", "士", "象", "馬", "車", "砲", "卒"]

class Piece(object):
    LABELS = [                                      # 棋子类型定义
        'GENERAL', 'ADVISOR', 'ELEPHANT', 'HORSE', 'CHARIOT', 'CANNON',
        'SOLDIER'
]

    SCORES = [0.99, 0.02, 0.02, 0.04, 0.09, 0.045, 0.01]    # 子力评估

    def __init__(self, _type, color='red'):
        self._type = _type                          # 棋子类型
        self.color = color                          # 红方/黑方
        self.rule = getattr(rule, self.label.lower())   # 规则对象

    def get_chinese_label(idx, color):
        if color == 'red':
            return red_labels[idx]
        else:
            return black_labels[idx]

    @property
    def label(self):
        return self.namemap[self._type]

    @property
    def score(self):                                    # 读取本棋子子力
        if self.color == 'red':
            return self.SCORES[self._type]
        else:
            return -self.SCORES[self._type]

    def is_enemy(self, another_piece):                  # 检查某棋子是否属于对方
        return self.color != another_piece.color

    def is_general(self):                               # 本棋子是否是“将”/“帅”
        return self._type == Piece.GENERAL

    def is_soldier(self):
        return self._type == Piece.SOLDIER              # 本棋子是否是“兵”/“卒”
```

上述代码用 Piece 类定义了一个象棋棋子，在构造函数中设定了该棋子的三个固定属性：棋子类型、棋子属于哪一方、行棋规则类。

2. 棋盘

棋盘状态用 Board 类描述，代码如下：

```
class Board(draw.DrawDelegate, event.MouseDelegate):
    gx = 8                                  # 棋盘的水平点数
gy = 9                                      # 棋盘的垂直点数

    def onInit(self):
        initpieces = {                      # 定义初始时刻的所有棋子
            (4, 9): Piece.GENERAL,
            (3, 9): Piece.ADVISOR,
            (5, 9): Piece.ADVISOR,
            (2, 9): Piece.ELEPHANT,
            (6, 9): Piece.ELEPHANT,
            (1, 9): Piece.HORSE,
            (7, 9): Piece.HORSE,
            (0, 9): Piece.CHARIOT,
            (8, 9): Piece.CHARIOT,
            (7, 7): Piece.CANNON,
            (1, 7): Piece.CANNON,
            (0, 6): Piece.SOLDIER,
            (2, 6): Piece.SOLDIER,
            (4, 6): Piece.SOLDIER,
            (6, 6): Piece.SOLDIER,
            (8, 6): Piece.SOLDIER,
        }
        for c, _type in initpieces.items():
            self.pieces[c] = Piece(_type, 'red')
            ix, iy = c
            d = (ix, -iy + self.gy)
            self.pieces[d] = Piece(_type, 'black')

    def to_array(self):                               # 将棋盘状态转换为数组
        board = np.zeros([self.gy + 1, self.gx + 1,
                      len(Piece.LABELS) * 2], np.uint8)
        board[:, :] = 0
        for c, piece in self.pieces.items():
```

```
            if piece.color == 'red':
                board[c[1], c[0], Piece.LABELS.index(piece.label)] = 1
            else:
                board[c[1], c[0],
                     Piece.LABELS.index(piece.label) + len(Piece.LABELS)] = 1
        return board

    def is_move_legal(self, old_lc, new_lc):    # 检查某步落子是否合法
        piece = self.pieces.get(old_lc)
        if piece is None:
            print("the move is illegal becaouse there is no piece in ", old_lc)
            return False
        if piece.color != self.turn:
            print("the move is illegal becaouse not in turn", old_lc,
                  piece.color)
            return False
        possible_moves = piece.rule(old_lc, self.pieces)
        ret = new_lc in possible_moves
        if not ret:
            print("the moves are impossible: ",
                  old_lc, new_lc, possible_moves)
        return ret

    def is_game_over(self):                     # 对局是否已经结束
        general_count = 0
        for c, piece in self.pieces.items():
            if piece.is_general():
                general_count += 1
        return general_count < 2

    def move_piece(self, old_lc, new_lc):       # 移动某个棋子
        p = self.pieces.pop(old_lc)
        if not p:
            return
        self.pieces[new_lc] = p
```

象棋的着法与围棋略有不同，在围棋中某次行棋只需给出“落子”点即可，而在象棋中由“提子/落子”两步完成。因此在 is_move_legal()、move_piece()等函数中，用两个坐标参数分别表达提子点和落子点，即 old_lc 和 new_lc。每个坐标参数都是一个二维 tuple，用两个整数分别表达棋盘横坐标和纵坐标。

3. 行棋规则

在棋子类Piece中有一个属性rule用于定义该棋子的行棋规则，其实它指向的是rule.py文件中的规则函数。如下是该文件中的一些示例代码：

```
def advisor(curr_pos, board_pieces):                        # “士”的规则
    ''' Rule for advisor. '''
    x, y = curr_pos

    def diagonal(x, y):
        ''' Diagonal posistions starting at (x,y) '''
        for nx in (-1, 1):
            for ny in (-1, 1):
                yield (x + nx, y + ny)

    candidates = diagonal(x, y)
    candidates = (c for c in candidates
                  if empty_or_enemy(curr_pos, c, board_pieces))
    candidates = (c for c in candidates if is_in_palace(c))

    possible_moves = list(candidates)
    return possible_moves

def soldier(curr_pos, board_pieces):                        # “兵”的规则
    x, y = curr_pos
    forward = get_forward(curr_pos, board_pieces)
    if in_enemy_territory(curr_pos, board_pieces):
        candidates = [
            (x + 1, y),
            (x - 1, y),
            (x, y + forward),
        ]
    else:
        candidates = [(x, y + forward)]
    candidates = [
        c for c in candidates if empty_or_enemy(curr_pos, c, board_pieces)
    ]
    candidates = [c for c in candidates if 0 <= c[0] <= int(widget.Board.gx)]
    candidates = [c for c in candidates if 0 <= c[1] <= int(widget.Board.gy)]
    return candidates
```

以上是象棋中两个最简单棋子“士”和“兵”的规则函数，每个规则函数的输入参数是棋子的当前坐标，返回的是该棋子的所有可能移动点。其他类型棋子的规则函数略显复杂，此处不再赘述，有兴趣的读者可翻阅源码。

10.5.3　左右互搏

Selfplay 是 AlphaGo Zero 自我学习的一个关键设计，在 Minigo 中这部分代码主要保存在 cchess.py、strategies.py 和 selfplay_mcts.py 三个模块中。

1. cchess.py

在 Xiangqi 模块中定义了中国象棋的所有静态盘面信息。在 Minigo 中，用 cchess.py 模块中定义的 Position 类对其进行封装，加入一些行棋动态属性，定义 MCTS、DDPG 等后续模块需要的接口，关键代码如下：

```
from widget import Piece, Board          # 引入 widget.py
import rule                              # 引入 rule.py

N_row = Board.gy + 1                     # Minigo 与 Xiangqi 中的坐标体系不同
N_column = Board.gx + 1

EMPTY_BOARD = Board(200)
EMPTY_BOARD.onInit()                     # 定义棋盘初始状态

class Position():
    def __init__(self,
                board=None,
                n=0,
                recent=tuple(),
                board_deltas=None,
                to_play=RED):
        # self.board 是一个内嵌的 widget.Board 对象
        self.board = board if board is not None
                        else copy.deepcopy(EMPTY_BOARD)
        self.n = n                                        # 已经下了几步棋
        self.recent = recent                              # 行棋记录
        self._to_play = to_play                           # 当前轮到谁走棋

    @property
```

```
    def to_play(self):
        return self._to_play

    @to_play.setter
    def to_play(self, to_play):                          # 设置 to_play
        self._to_play = to_play
        self.board.turn = to_play == RED and "red" or "black"

    def is_move_legal(self, move):                       # 行棋是否合法
        if move is None:
            return False
        move = (cc_coords.swap_coord(move[0]),
            cc_coords.swap_coord(move[1]))
        return self.board.is_move_legal(move[0], move[1])

    def all_legal_moves(self):                           # 查找所有可能行棋
        legal_moves = np.zeros([N_row * N_column * N_row * N_column], np.int8)
        for c, piece in self.board.pieces.items():
            if piece.color != self.board.turn:
                continue
            possible_news = piece.rule(c, self.board.pieces)
            c = cc_coords.swap_coord(c)
            for new_c in possible_news:
                new_c = cc_coords.swap_coord(new_c)
                legal_moves[cc_coords.to_flat((c, new_c))] = 1
        if len(legal_moves) <= 0:
            print(self.board.turn, " has no possible moves!")
        return legal_moves

    def move_piece(self, move):                          # 行棋
        self.board.move_piece(
            cc_coords.swap_coord(move[0]), cc_coords.swap_coord(move[1]))

    def play_move(self, move, color=None, mutate=False):   # 行棋并记录
        if color is None:
            color = self.to_play

        pos = self if mutate else copy.deepcopy(self)

        if not self.is_move_legal(move):         # 安全检查
            raise IllegalMove("{} move at {} is illegal: \n{}".format(
```

```
                "Black" if self.to_play == BLACK else "Red",
                cc_coords.to_kgs(move), self))

        pos.move_piece(move)

        pos.n += 1
        pos.recent += (PlayerMove(color, move, pos.score() - self.score()), )
        pos.to_play = pos.to_play * -1          # 行棋后轮换
        return pos

    def is_game_over(self):
        return self.board.is_game_over()

    def score(self):                            # 当前盘面双方的子力差距
        score = 0
        for c, piece in self.board.pieces.items():
            s = piece.score
            if piece.is_soldier() and rule.in_enemy_territory(
                    c, self.board.pieces):
                s *= 2                          # 已过河的“兵”子力加倍
            score += s
        return score

    def result(self):                           # 用子力差计算行棋价值
        score = self.score()
        if score > 0.0001:
            return 1
        elif score < -0.0001:
            return -1
        else:
            return 0
```

代码逻辑比较简单，有几点值得注意的是：

◎ Position 与 Board 的棋盘坐标系正好相反，即 Position 中（x, y）用于表达第 x 行、第 y 列，而 Board 中意义为第 y 行、第 x 列；在 Minigo 中用 cc_coords 里的工具函数完成坐标转换。

◎ 在 Position 中用一个变量 move 代表一次行棋，move 对象是对 Board 中 old_rc 和 new_lc 的一个 tuple 封装。

◎ Position.score()用于计算当前盘面的子力差距，正数表示红方占优、负数表示黑方占优。

◎ Position.all_legal_moves()返回的是一个数组，该数组是对所有可能的 move 进行扁平化（flatten）后的形式，数组中元素为 1 表示该 move 合法。

◎ Position.result()函数在对局结束时被使用，此时子力差距占优的一方获得全部价值（即强化学习中的 Value）。

2. strategies.py

在 strategies.py 中定义的 MCTSPlayer 类可以看成一个对弈者，它使用 MCTS、DDPG 两个工具在 Position 定义的盘面上寻找最佳行动，关键代码如下：

```
class MCTSPlayer(MCTSPlayerInterface):
    def __init__(self,
                 network,                  # DDPG 神经网络
                 seconds_per_move=5,
                 num_readouts=0,           # MCTS 的并行处理属性
                 resign_threshold=None,
                 verbosity=0,              # 日志级别
                 two_player_mode=False,    # 自我学习时设置 False
                 timed_match=False):
        # 此处省略属性初始化代码
        pass

    def get_position(self):
        return self.root.position if self.root else None

    def get_root(self):                    # 当前 MCTS 的根结点
        return self.root

    def suggest_move(self, position):
        start = time.time()

        if self.timed_match:
            while time.time() - start < self.seconds_per_move:   # 限时搜索
                self.tree_search()                  # 执行 MCTS 搜索
        else:
            current_readouts = self.root.N
```

```
        while self.root.N < current_readouts + self.num_readouts:
            self.tree_search()                   # 执行 MCTS 搜索

        return self.pick_move()                  # 选取 Action

    def play_move(self, c):                      # 执行某个 Action
        if not self.two_player_mode:
            self.searches_pi.append(             # 记录本次行棋的 Policy 分布
                self.root.children_as_pi(
                    self.root.position.n <= self.temp_threshold))
        self.qs.append(self.root.Q)              # 记录当前盘面的 Q 值
        self.comments.append(self.root.describe())
        try:
            # 行棋
            self.root = self.root.maybe_add_child(cc_coords.to_flat(c))
        except cchess.IllegalMove:
            print("Illegal move")                # 如果是非法行棋则回退
            if not self.two_player_mode:
                self.searches_pi.pop()
            self.qs.pop()
            self.comments.pop()
            return False
        self.position = self.root.position
        self.position.current = self.root.position
        del self.root.parent.children
        return True

    def pick_move(self):                         # 行棋
        if self.root.position.n >= self.temp_threshold:
            # 选择被采样次数最多的 MCTS 中的子结点作为 Action
            fcoord = np.argmax(self.root.child_N)
        else:
            # 在采样次数较多的几个 Action 中随机选择一个
            cdf = self.root.child_N.cumsum()
            cdf /= cdf[-2]
            selection = random.random()
            fcoord = cdf.searchsorted(selection)
        return cc_coords.from_flat(fcoord)

    def tree_search(self, parallel_readouts=None):      # 搜索 MCTS
        if parallel_readouts is None:
```

```
            parallel_readouts = FLAGS.parallel_readouts
        leaves = []
        failsafe = 0
        while len(leaves
                ) < parallel_readouts and failsafe < parallel_readouts * 2:
            failsafe += 1
            leaf = self.root.select_leaf()       # MCTS 的 Selection 操作
            if leaf.is_done():
                value = 1 if leaf.position.score() > 0 else -1
                leaf.backup_value(value, up_to=self.root)
                continue
            leaf.add_virtual_loss(up_to=self.root)
            leaves.append(leaf)
        if leaves:
            # 从 DDPG 网络预测本次 Selection 的 Policy 和 Value
            move_probs, values = self.network.run_many(
                [leaf.position for leaf in leaves])
            for leaf, move_prob, value in zip(leaves, move_probs, values):
                leaf.revert_virtual_loss(up_to=self.root)
                # MCTS 的 Expansion、Simulation 和 Backpropagation 操作
                leaf.incorporate_results(move_prob, value, up_to=self.root)
        return leaves

    def is_done(self):                           # 是否对局已结束
        return self.result != 0 or self.root.is_done()

    def should_resign(self):                     # 是否投子认输
        if self.root.position.to_play > 0:       # 如果子力差距大于 0.2 则认输
            return self.root.position.score() < -0.2
        else:
            return self.root.position.score() > 0.2
```

上述注释中的 Action 是一个整数，是对 Position 中定义的 move 扁平化后的值。这部分代码核心在于 tree_search()函数，其中给出了 MCTS 与 DDPG 的结合方式。即：

◎ 用 MCTS 采样叶子结点。

◎ 用 DDPG 预测该叶子结点的 Policy 与 Value。

◎ 将 DDPG 的输出作为该叶子结点进一步 Expansion、Simulation、Backpropagation 的依据。

3. selfplay_mcts.py

模块 selfplay_mcts.py 可以看成 Selfplay 功能的主程序，其调用 MCTSPlayer 实现红棋、黑棋的轮番对弈，关键代码为：

```
def play(network, verbosity=0):
    readouts = flags.FLAGS.num_readouts

    player = MCTSPlayer(                                # 初始化 MCTSPlayer
        network, verbosity=verbosity, resign_threshold=resign_threshold)

    player.initialize_game()

    # 初始状态的 MCTS 搜索与扩展
    first_node = player.root.select_leaf()
    prob, val = network.run(first_node.position)
    first_node.incorporate_results(prob, val, first_node)

    while True:
        start = time.time()
        player.root.inject_noise()
        current_readouts = player.root.N
        # 大量的 MCTS 采样
        while player.root.N < current_readouts + readouts:
            player.tree_search()

        if player.should_resign():                      # 检查是否需要认输
            player.set_result(
                -1 * player.root.position.to_play, was_resign=True)
            break

        move = player.pick_move()                       # 选择一个 Action
        player.play_move(move)                          # 执行 Action
        if player.root.is_done():
            player.set_result(player.root.position.result(),
                was_resign=False)
            break
    return player
```

上述代码非常简单，即在一个循环里不断地执行 MCTS 搜索、选择和执行 Action。因为在 Position.play_move()函数中每次执行 Action 后会重新设置 to_play 属性轮换行棋角色，

因此在上述循环中也就是一个左右互搏的过程。

10.5.4 MCTS 详解

在上节代码中，关联最多的一个模块就是 MCTS，这也是 AlphaGo 的核心技术之一。但这部分代码其实并不复杂，完全封装在 mcts.py 的 MCTSNode 类中，该类定义的是蒙特卡洛搜索树中的一个结点，关键代码如下：

```
class MCTSNode(object):
    def __init__(self, position, fmove=None, parent=None, player=None):
        self.player = player                        # MCTSPlayer 对象
        self.parent = parent                        # 父结点
        self.fmove = fmove                          # 进入本结点的 Action
        self.position = position                    # Position 对象
        self.is_expanded = False                # 本结点是否已经执行 Expended 操作
        # 在该数组中，用 0 值代表一个合法 Action；用一个很大的值表示非法 Action
        self.illegal_moves = 1000000000 * (1 -
                    self.position.all_legal_moves())
        self.child_N = np.zeros(                    # 所有子结点的访问量
            [cchess.N_row * cchess.N_column * cchess.N_row *
                    cchess.N_column],
            dtype=np.float32)
        self.child_W = np.zeros(                    # 所有子结点的权重
            [cchess.N_row * cchess.N_column * cchess.N_row *
                    cchess.N_column],
            dtype=np.float32)
        self.original_prior = np.zeros(             # DDPG 输出的 Policy
            [cchess.N_row * cchess.N_column * cchess.N_row *
                    cchess.N_column],
            dtype=np.float32)
        self.child_prior = np.zeros(                # 加噪声后的 DDPG Policy
            [cchess.N_row * cchess.N_column * cchess.N_row *
                    cchess.N_column],
            dtype=np.float32)
        self.children = {}                          # 所有扩展过的子结点
        return

    @property
    def child_action_score(self):                   # 所有子结点的打分数组
        return self.child_Q * self.position.to_play - self.illegal_moves +
```

```
self.child_U

        @property
        def child_Q(self):                                # 所有子结点的 Q 值
            return self.child_W / (1 + self.child_N)

        @property
        def child_U(self):                                # 子结点的 U 值
            return (FLAGS.c_puct * math.sqrt(1 + self.N) * self.child_prior /
                    (1 + self.child_N))

        @property
        def Q(self):
            return self.W / (1 + self.N)

        @property
        def N(self):
            return self.parent.child_N[self.fmove]

        @N.setter
        def N(self, value):
            self.parent.child_N[self.fmove] = value

        @property
        def W(self):
            return self.parent.child_W[self.fmove]

        @W.setter
        def W(self, value):
            self.parent.child_W[self.fmove] = value

        @property
        def Q_perspective(self):
            return self.Q * self.position.to_play

        def select_leaf(self):                            # Selection 操作
            current = self
            while True:
                current.N += 1                            # 累加访问数
                if not current.is_expanded:               # 找到了叶子结点
                    break
```

```
            best_move = np.argmax(current.child_action_score)# 最佳子结点
            current = current.maybe_add_child(best_move)     # 深度搜索
        return current

    def maybe_add_child(self, fcoord):      # 返回指定子结点，即深度搜索
        if fcoord not in self.children:     # 如果本 Action 尚未被搜索，则新建
            new_position = self.position.play_move(
                                 cc_coords.from_flat(fcoord))
            self.children[fcoord] = MCTSNode(
                new_position, fmove=fcoord, parent=self, player=self.player)
        return self.children[fcoord]

    def add_virtual_loss(self, up_to):      # 对搜索过的叶子增加 virtual_loss
        loss = self.position.to_play * self.position.n
        self.virtual_losses.append(loss)
        self.W += loss
        if self.parent is None or self is up_to:
            return
        self.parent.add_virtual_loss(up_to)

    def revert_virtual_loss(self, up_to):      # 恢复 virtual_loss
        revert = -self.virtual_losses.pop()
        self.W += revert
        if self.parent is None or self is up_to:
            return
        self.parent.revert_virtual_loss(up_to)

    # MCTS 的三步关键操作
    def incorporate_results(self, move_probabilities, value, up_to):
        self.is_expanded = True
        # 设置子结点的 policy，即 Expension
        self.original_prior = self.child_prior = move_probabilities

        # 所有子结点的初始 Value 与当前 Value 相同
        self.child_W = np.ones(
            [cchess.N_row * cchess.N_column * cchess.N_row *
                  cchess.N_column],
            dtype=np.float32) * value
        # Simulation 和 Backpropagation
        self.backup_value(value, up_to=up_to)
```

```
    def backup_value(self, value, up_to):
        self.W += value                    # 累加本结点的 Value
        if self.parent is None or self is up_to or type(
                self.parent) is DummyNode:
            return
        # 通过父结点向上 Backpropagation
        self.parent.backup_value(value, up_to)

    def is_done(self):
        return self.position.is_game_over(
        ) or self.position.n >= FLAGS.max_game_length

    def inject_noise(self):                # 为 Policy 增加噪声，增加样本多样性
        dirch = np.random.dirichlet(
            [FLAGS.dirichlet_noise_alpha] *
            (cchess.N_row * cchess.N_column * cchess.N_row *
                   cchess.N_column))
        self.child_prior = (
            self.child_prior * (1 - FLAGS.dirichlet_noise_weight) +
            dirch * FLAGS.dirichlet_noise_weight)

    def children_as_pi(self, squash=False):# 在一批采样后，返回 Policy
        probs = self.child_N
        legal_moves = self.position.all_legal_moves()
        # 用 maximum()防止合法 Action 在 Policy 中被设为零
        probs = np.maximum(probs, legal_moves)
        if squash:
            probs = probs**.98
        return probs / np.sum(probs)
```

MCTS 的关键步骤在注释中说得比较明白，如 select_leaf ()函数、incorporate_results()函数和 backup_value()函数。对这些函数中用到的几个 MCTS 属性说明如下。

◎ W：是结点的累计 Value，即自己状态的 Value 和所有探索过子结点的 Value 之和。

◎ N：是结点的访问数量，结点每次被“路过”一次，该值加一。

◎ Q：是结点的平均 Value，即 $W/(N+1)$，也就是强化学习中的 Q 值。

◎ U：$U(s,a) \propto P(s,a)/(1+N(s,a))$，该值用于引导 MCTS 向 Policy 较大的子结点进行深度搜索；但随着采样数量的增加，Policy 的影响逐渐减小，使得 MCTS 有更

多的机会进行宽度搜索。

◎ child_action_score：综合考虑 Q 值和 U 值，并排除 illegal_moves 中的 Action。该值是 MCTS 每次 Select 子结点的最终依据。

此外，add_virtual_loss()和 revert_virtual_loss()这一对函数的目的需要结合 MCTSPlayer.tree_search()函数理解。在该函数中，每选取一个子结点都会用 add_virtual_loss()增加 virtual_loss，在扩展该子结点之前减少 virtual_loss。这些 virtual_loss 最终都被累加到相关结点的 W 值中，因此它的作用是在 tree_search()的多次采样中尽量避免搜索同一条深度路径，因此也是一种增加宽度搜索的机制。

注意：一次 tree_search()中采样的数量由 MCTS 的超参数 parallel_readouts 控制。

10.5.5 DDPG 详解

在 Minigo 中，神经网络作为一种训练/保存 Policy 和 Value 的模型而使用，即应用了 DDPG 强化学习方法。该神经网络其实可以有任何内部结构，只需要满足如下两个条件。

◎ 使用当前环境状态作为输入层：棋盘状态+行棋角色。

◎ 使用 Policy+Value 作为输出层。

1. DualNetwork 对象

神经网络的完整代码保存在 dual_net.py 中，其中 DualNetwork 是在 MCTSPlayer 中使用的 network 参数对象：

```
class DualNetwork():
    def __init__(self, save_file):
        self.sess = tf.Session(graph=tf.Graph(), config=config) # 初始化
        self.initialize_graph()                    # 建立 TensorFlow 图结构

    def initialize_graph(self):
        with self.sess.graph.as_default():
            # features 是输入层，labels 是输出层
            features, labels = get_inference_input()
            # 用 model_fn 函数建立 Graph
            estimator_spec = model_fn(features, labels,
                                  tf.estimator.ModeKeys.PREDICT)
```

```
            self.inference_input = features
            self.inference_output = estimator_spec.predictions
            if self.save_file is not None:
                # 从已有文件中读取网络参数
                self.initialize_weights(self.save_file)
            else:
                # 初始化一组新的网络参数
                self.sess.run(tf.global_variables_initializer())

    # run() 一个样本数据
    def run(self, position, use_random_symmetry=False):
        probs, values = self.run_many(
            [position], use_random_symmetry=use_random_symmetry)
        return probs[0], values[0]

    # run() 多个样本数据
    def run_many(self, positions, use_random_symmetry=False):
        processed = list(map(features_lib.extract_features, positions))
        outputs = self.sess.run(                          # 执行会话
            self.inference_output,
            feed_dict={self.inference_input: processed})
        probabilities, value = outputs['policy_output'], outputs[
            'value_output']                               # Policy 与 Value
        return probabilities, value
```

在 DualNetwork 中清晰地定义了 TensorFlow 图建立、执行的接口函数。其中通过 get_inference_input()函数进一步定义了输入层与输出层的数据格式，通过 model_fn()定义了详细的图结构。

2. 输入/输出层数据

在 get_inference_input()定义了输入层与输出层的数据格式：

```
def get_inference_input():
    return (tf.placeholder(
        tf.float32, [
            None, cchess.N_row, cchess.N_column,
            features_lib.NEW_FEATURES_PLANES
        ],
        name='pos_tensor'), {
            'pi_tensor':
```

```
            tf.placeholder(tf.float32, [
                None,
                cchess.N_row * cchess.N_column * cchess.N_row *
                    cchess.N_column
            ]),
            'value_tensor':
            tf.placeholder(tf.float32, [None])
    })
```

其中，输出层的（'pi_tensor'，'value_tensor'）已经被多次强调，即强化学习的 Policy 和 Value。而输入层是一个形如[None，cchess.N_row，cchess.N_column，features_lib.NEW_FEATURES_PLANES]的四阶张量。第一阶是用于以 batch 形式批次传入样本，后面三阶才是每条样本真正的特征数据。

那么这些特征数据是什么呢？它们都被定义在了 features.py 文件中，关键代码如下：

```
@planes(7 * 2)                         # 每一方有 7 种棋子，因此双方共 7×2 类棋子
def piece_features(position):
    ret = position.board.to_array()
    return ret

@planes(1)                             # 用一个整数定义当前的行棋方
def cchess_color_to_play_feature(position):
    if position.to_play == cchess.RED:  # 红棋走为 1
        return np.ones([cchess.N_row, cchess.N_column, 1], dtype=np.uint8)
    else:                              # 黑棋走为 0
        return np.zeros([cchess.N_row, cchess.N_column, 1], dtype=np.uint8)

NEW_FEATURES = [piece_features, cchess_color_to_play_feature]

NEW_FEATURES_PLANES = sum(f.planes for f in NEW_FEATURES)
```

即输入层是一个形如[cchess.N_row，cchess.N_column，15]的数据，定义了棋盘上每一个位置的状态和当前的行棋方。

3. Graph 结构

在 model_fn()函数中定义了深度学习网络结构：

```
def model_fn(features, labels, mode):
    my_batchn = functools.partial(                 # Batch Norm 算子
```

```
    tf.layers.batch_normalization,
    momentum=.997,
    epsilon=1e-5,
    fused=True,
    center=True,
    scale=True,
    training=(mode == tf.estimator.ModeKeys.TRAIN))

my_conv2d = functools.partial(                   # 2D 卷积层
    tf.layers.conv2d,
    filters=FLAGS.conv_width,
    kernel_size=[3, 3],
    padding="same")

def my_res_layer(inputs):                        # 带 ResNet 的卷积层
    int_layer1 = my_batchn(my_conv2d(inputs))
    initial_output = tf.nn.relu(int_layer1)
    int_layer2 = my_batchn(my_conv2d(initial_output))
    output = tf.nn.relu(inputs + int_layer2)
    return output

# 接收输入张量
initial_output = tf.nn.relu(my_batchn(my_conv2d(features)))

shared_output = initial_output
for _ in range(FLAGS.trunk_layers):     # 用 ResNet 建立多个中间卷积隐藏层
    shared_output = my_res_layer(shared_output)

# Policy head
policy_conv = tf.nn.relu(                        # Policy 卷积层
    my_batchn(
        my_conv2d(shared_output, filters=2, kernel_size=[1, 1]),
        center=False,
        scale=False))
logits = tf.layers.dense(                        # Policy 全连接层
    tf.reshape(policy_conv, [-1, cchess.N_row * cchess.N_column * 2]),
    cchess.N_row * cchess.N_column * cchess.N_row * cchess.N_column)

# Policy 的 Softmax 输出层
policy_output = tf.nn.softmax(logits, name='policy_output')
```

```
value_conv = tf.nn.relu(                               # Value 卷积层
    my_batchn(
        my_conv2d(shared_output, filters=1, kernel_size=[1, 1]),
        center=False,
        scale=False))
value_fc_hidden = tf.nn.relu(                          # Value 全连接层
    tf.layers.dense(
        tf.reshape(value_conv, [-1, cchess.N_row * cchess.N_column]),
        FLAGS.fc_width))
value_output = tf.nn.tanh(                             # Value 输出层
    tf.reshape(tf.layers.dense(value_fc_hidden, 1), [-1]),
    name='value_output')

############### 这里开始是训练中需要的损失定义与优化器定义 ##########
global_step = tf.train.get_or_create_global_step()
policy_cost = tf.reduce_mean(                          # 定义 Policy 损失计算方式
    tf.nn.softmax_cross_entropy_with_logits_v2(
        logits=logits, labels=tf.stop_gradient(labels['pi_tensor'])))
value_cost = tf.reduce_mean(                           # 定义 Value 损失计算方式
    tf.square(value_output - labels['value_tensor']))
# 这里省略若干其他统计量的定义
with tf.control_dependencies(update_ops):
    train_op = tf.train.MomentumOptimizer(learning_rate,# 定义优化器
                                          FLAGS.sgd_momentum).minimize(
                                              combined_cost,
                                              global_step=global_step)
metric_ops = {
    'policy_cost':
    tf.metrics.mean(policy_cost),
    'value_cost':
    tf.metrics.mean(value_cost),
    # 此处省略若干其他 metrics
}

for metric_name, metric_op in metric_ops.items():
    # 加入 tensorboard 统计图
    tf.summary.scalar(metric_name, metric_op[1])

return tf.estimator.EstimatorSpec(  # 使用 estimator 执行上述图结构
```

```
        mode=mode,                          # 是否是训练模式
        predictions={                       # 输出层
            'policy_output': policy_output,
            'value_output': value_output,
        },
        loss=combined_cost,                 # 损失定义
        train_op=train_op,                  # 优化器
        eval_metric_ops=metric_ops,         # 统计 metrics
    )
```

代码中的注释详细解析了模型的整体设计框架，该网络基本上用到了所有第 9 章中学过的当前深度学习网络的重要技术：CNN、Batch Norm、ResNet 等。其中省略了部分非关键细节，有兴趣的读者可以参考本书源码。

10.5.6　运行展示：训练

因为篇幅原因，本节不再分析 Minigo 的更多细节代码。相信读者此时已经跃跃欲试，希望能动手实践该项目。接下来依次介绍它的训练调用、对局展示、人机博弈接口等。

Minigo 的所有人机交互接口定义在项目的 main.py 文件中。可以通过三条命令开始训练自己的 Minigo 模型。首先初始化一个神经网络模型：

```
# python3 main.py bootstrap --working-dir=./tmp/minigo_base
   --model-save-path=./tmp/minigo_base/models/000000-bootstrap
```

上述代码在--working-dir 所指定的目录中建立一个在 dual_net.py 中定义的 TensorFlow 模型文件。然后，通过如下命令开始进行 Selfplay（自我对决）：

```
# python3.6 main.py multi-selfplay 10000 ./tmp/minigo_base
   --num_readouts=1000 -v 3
```

上述命令执行 10000 局的中国象棋 Selfplay，对弈结果作为 TensorFlow 数据文件保存在./tmp/minigo_base/data/selfplayxxxx/local_work 目录中。其中的参数--num_readouts 用于定义每走一步棋需要执行的蒙特卡洛搜索采样次数。

Selfplay 命令的执行非常耗时，执行过程每走一步棋会输出如图 10-19 所示的界面。

```
move         : action     Q      U      P    P-Dir    N  soft-N  p-delta  p-rel
(7,1-->7,4) :  0.1966637459952  0.0852379128337 0.1114258331615 0.2666445970535 0.3555261492729    33 0.3173  0.05066  0.19
(7,7-->7,4) :  0.1992554978934  0.0752017498016 0.1240537480918 0.2357445359230 0.3143260478973    26 0.2500  0.01426  0.06
(9,0-->8,0) :  0.1617137101874  0.1505163908005 0.0111973193869 0.0165501274168 0.0220668371767    20 0.1923  0.17576  10.62
(9,1-->7,2) :  0.0799947470009  0.0418231710792 0.0381715759217 0.0134331565350 0.0179108753800     4 0.0385  0.02503  1.86
(9,7-->7,6) :  0.0247033764539 -0.0255826357752 0.0502860122291 0.0176964104176 0.0235952138901     4 0.0385  0.02077  1.17
(9,8-->8,8) :  0.0316933019880 -0.0187414474785 0.0504347494665 0.0141990026459 0.0189320035279     3 0.0288  0.01465  1.03
(6,0-->5,0) :  0.0051865712535 -0.0781592950225 0.0833458662760 0.0234645395941 0.0312854498625     3 0.0288  0.00538  0.23
(6,8-->5,8) :  0.0842302981493  0.0115893287584 0.0726409693909 0.0153380753472 0.0204507671297     2 0.0192  0.00389  0.25
(9,1-->7,0) :  0.0793226827787  0.0040289610624 0.0752937217162 0.0158982016146 0.0211976021528     2 0.0192  0.00333  0.21
(9,7-->7,8) :  0.0152023413919 -0.0864518955350 0.1016542369269 0.0214642006904 0.0286189336330     2 0.0192 -0.00223 -0.10
(7,7-->7,6) : -0.1146238591701 -0.1289135068655 0.0142896476954 0.0030172462575 0.0040229950100     2 0.0192  0.01621  5.37
(7,7-->7,8) :  0.0588341029900  0.0127815892920 0.0460525136980 0.0064826312291 0.0004588768643     1 0.0096  0.00313  0.48
(9,3-->8,4) :  0.0553210226425  0.0099764931947 0.0453445294478 0.0063829710707 0.0085106277838     1 0.0096  0.00323  0.51
(9,5-->8,4) :  0.0111580443536 -0.0361140333116 0.0472720776652 0.0066543044522 0.0088724056259     1 0.0096  0.00296  0.44
50 possible moves:
Q: 0.07520
1: 100 readouts, 2.760 s/100. (2.76 sec)
Played >> (7,7-->7,4)
   0 1 2 3 4 5 6 7 8
0  車馬象士將士象馬車
1  . . . . . . . . .
2  . 砲. . . . . 砲.
3  卒. 卒. 卒. 卒. 卒
4  . . . . . . . . .
5  . . . . . . . . .
6  兵. 兵. 兵. 兵. 兵
7  . 炮. . 炮. . * .
8  . . . . . . . . .
9  俥傌相仕帅仕相傌俥
 turn is black, balance 0.000
```

图 10-19　Selfplay 命令的执行过程

图 10-19 中上半部分的表格记录了本次 MCTS 所搜索过的 Action 和它们在 mcts.py 中定义的各属性最终值。比如在图 10-19 中，第一条记录内容如下。

◎ “提子（7, 1），落子到（7, 4）”，每颗棋子的坐标如图中 10-19 下半部分所示。

◎ 记录中的几列数字分别是：child_action_score 值、*Q* 值、*U* 值、child_prior、original_prior、*N* 值等。

在表格中越是上方的 Action 越容易成为被 MCTS 推荐的行棋方式，但由于在 MCTSPlayer.pick_move()函数中加入了随机因素，因此最终也可能选择其他行棋方式。比如图 10-19 中就选择了排第二的（7, 7 →7, 4）作为 played action。

由于在启动命令时指定了执行 10000 次的 Selfplay，该命令在短期内不会退出。但当通过输出发现一次对弈已经完成时，可以通过下述命令开始神经网络的训练：

```
# python3 main.py multi-train-dir 99999
      ./tmp/minigo_base ./tmp/minigo_base/models/000001-first
```

该命令在./tmp/minigo_base 的所有子目录中查找可以使用的 Selfplay 样本文件，并加载它们训练 DualNetwork 网络，训练结果保存在./tmp/minigo_base/models/000001-first 中。

注意：*在至少一个完整的 Selfplay 对局结束后才能启动 train 进程。*

由于 train 命令也给出了一个很高的执行次数（99999 次），它也不会在短期内退出。因此，形成了如图 10-20 所示的 Selfplay 和 train 两个进程相互利用、逐步提高的过程。

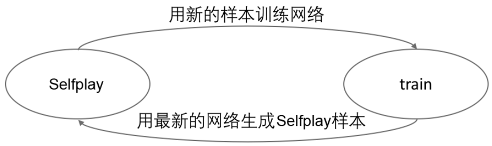


图 10-20　Selfplay 与 train 的相互作用

保持图中两个进程的运行状态，即可在./tmp/minigo_base 中训练出越来越强的中国象棋博弈机器人。

10.5.7　运行展示：查看统计

本项目在深度网络训练过程中也输出了比较丰富的优化统计数据，可以用如下命令启动 tensorboard：

```
# python3 -m tensorboard.main --logdir ./tmp/minigo_base/
```

打开浏览器访问 http://127.0.0.1:6006/，可以观察到图结构和统计图表，如图 10-21 所示是约 1000 次对局后的损失统计。

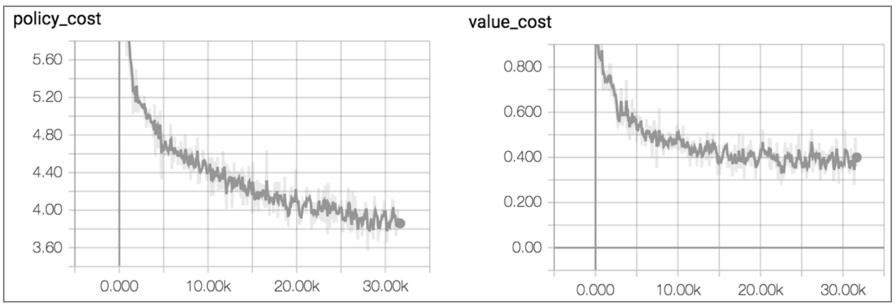


图 10-21　DDPG 的输出损失统计

可以发现 Policy 仍处于下降过程中，这说明此时 Minigo 对象棋规则的掌握仍然不够，随着 Selfplay 和 train 的进行，对弈能力仍有提高空间。

10.5.8　运行展示：当头炮、把马跳

在图 10-20 中循环训练的初期，Minigo 走出的多数是一些无意义的落子，比如对局初期就开始出边兵、上将等。随着训练的进行，Minigo 会逐步倾向于吃子、攻击对方“帅/将”。在 Selfplay 生成了 1000 次左右对弈后的某次对弈初期的八步棋如图 10-22 所示。

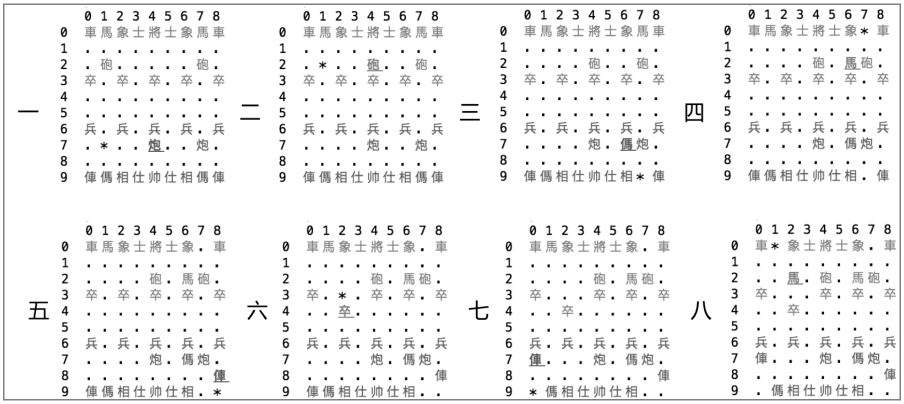


图 10-22　中国象棋 Minigo 自我博弈展示

从图 10-22 中可以发现，在代码中没有给出任何具体行棋指示的情况下，经过一段时间的训练后 Minigo 竟然学会了“当头炮、把马跳”这样大多数象棋爱好者谙熟于心的经典开局。并且在随后的几步落子中，没有明显的废棋落子。

10.5.9　运行展示：人机博弈

通过如下命令可以开始用文本方式与自己训练的 Minigo 进行人机博弈：

```
# python3 main.py gtp -l ./tmp/minigo_base/models/000001-first
    --num_readouts=1000 -v 3
GTP engine ready
```

上述命令在输出“GTP engine ready”后说明已经进入了人机博弈状态。此时可以通过在控制台中输入如下三条命令之一进行对弈操作。

◎　showboard：显示当前棋盘状态。

◎　genmove：由 Minigo 机器人行棋。

◎　play（old_x，oldy_y-->new_x，new_y）：操作者行棋，通过参数后接的参数设定如何走棋。比如（0,1-->2,2）的含义是“位于（0, 1）处的棋子移动到（2, 2）位置”，参考图 10-22 中的坐标，该步棋是后手方跳马。

人机博弈对上述命令的响应与图 10-19 中的内容类似，此处不再进一步举例。读者在尝试后可能会发现打败自我学习了 1000 局左右的 Minigo 非常容易，继续增强该机器人可能的途径如下。

◎ 继续执行图 10-20 中的 Selfplay 和 train 进程，以期达到更高水平。

◎ 改善 DDPG 中对输入、输出数据的建模，减少特征与 Policy 的维度数。

◎ 重新设计 DDPG 中卷积网络的参数，当前使用的固定[3×3]卷积核更适合于围棋、五子棋等局部区域计算非常重要的棋类。

◎ 在训练中加入验证机制。

总之，开发一个足以实战的博弈机器人仍然需要很多的计算资源与细节设计。作为强化学习的教程案例本书不再深入，留给有兴趣的读者自行研究。

10.6　本章内容回顾

◎ 通过 AlphaGo 与 DeepBlue 的比较可以了解强化学习给传统人工智能注入了新的发展活力。

◎ 实践强化学习之前必须理解的几个要素与术语：Agent、Environment、Value、Policy、Reward、Action。

◎ 总体上强化学习中的相关算法可以分为三类：基于价值的算法、基于策略的算法、基于模型的算法。其中前两者在不同环境中具有通用性。

◎ 传统上基于价值的算法有：Q-Learning、Sarsa 等；在深度强化学习中则演化出了 DQN 及若干改进。

◎ 基于策略的算法主要是 Policy Gradient，在深度强化学习中相对应的是 DPN。

◎ 把基于价值和基于策略两种方式结合起来，形成了 Actor-Critic 算法，在深度强化学习中则是 DDPG。

◎ 问题搜索是传统人工智能的重要模型与算法，一般来说深度搜索更快，宽度搜索效果更好。

◎ 最小最大算法、Alpha-Beta 剪枝等能够寻找到博弈中的最佳决策，但往往代价太高无法实际应用。

◎ 蒙特卡洛搜索树（MCTS）是一种结合了深度搜索、宽度搜索的博弈算法。

◎ 目前棋类博弈机器人使用的核心技术架构是：MCTS+强化学习。

11

第 11 章 模型迁移

这是本书的最后一章，应该也是最容易阅读的一章。本章不涉及新的数学概念、算法或模型，而是用简短篇幅介绍谷歌和苹果公司发布的一些通用模型，启发读者快速将这些已有模型应用到自己的项目中。本章主要内容如下。

◎ 走向移动端：介绍将 TensorFlow 模型移植到 Android 和 iOS 的开发工具。

◎ 迁移学习：基于已有模型训练新模型的技术，解析一个迁移学习的小案例。

◎ 案例实战：基于 TensorFlow Hub 的迁移学习开发。

11.1 走向移动端

同源于谷歌公司，Android 无疑是 TensorFlow 生态圈中重要的组成部分，而机器学习

迁移库 CoreML 也是苹果公司在 2017 年发布的 iOS 11 的重要新特性。

由于在代码层面，移动端 App 使用各自的编程语言（Android 中主要是 Java，iOS 中是 Swift 和 Objective-C），这些超出了本书的主要目标，因此本节着重展示移动领域已有的成果和关键技术，有这方面项目需求的读者可以根据给出的资源自行实践。

11.1.1　Android 上的 TensorFlow

随着移动设备硬件的普及及网络带宽的发展，很多都市人成了手不离机的“低头族”。在移动端开发各种智能 App 成为了趋势，这其中当然包括机器学习 App。

相对于普通桌面应用，移动 App 似乎更是一个快消品，很多 App 的成功依赖于创始人敏锐的市场洞察力、快速组织与开发执行力。在 2017 年，谷歌和苹果都在他们的移动操作系统中增强了对机器学习技术的支持，对于移动开发者来说，调研和尝试他们发布的一些已经训练成型的案例模型是将机器学习技术应用到自己 App 中的第一步。

1. 可用模型

谷歌在 https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android 项目中维护了一组成熟的深度学习模型及开发指导。

◎　TF Classify：即图像处理中最基本的任务——图像识别分类。

◎　TF Detect：图像物件检测，使用了速度非常快的 SSD 架构。

◎　TF Stylize：趣味图像风格重构。

◎　TF Speech：简单语音识别。

这些模型的特点是没有特定的领域局限性，开发者自己训练往往非常耗时，因此适合拿来即用。

2. 体验

对于有经验的开发者来说，体验这些模型非常容易，只需要在下载源码后按照 Readme 中的步骤作细微配置调整后，即可用 Android Studio（Android 集成开发工具）编译并运行该项目。图 11-1 是官方发布的体验截图。

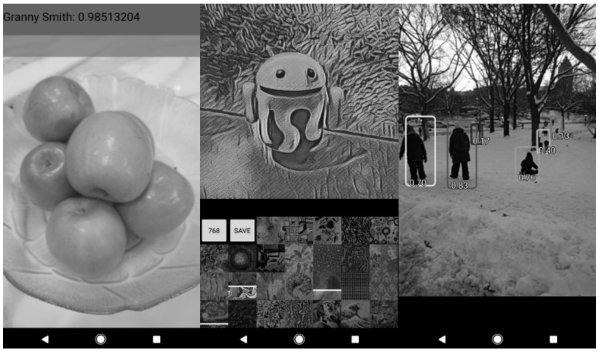


图 11-1　Android 图像模型体验（来自 GitHub 上本项目主页）

图 11-1 中从左到右分别是 TF Classify、TF Stylize、TF Detect 的体验效果。

3. 开发简述

上述的几类 Android TensorFlow 应用，其实都只是在使用特定模型的“预测”功能，即给深度学习输入图片，然后从输出层读取所要结果。因此它们都有统一的编程接口，这就是 org.tensorflow.contrib.android.TensorFlowInferenceInterface，使用它在移动端开发机器学习应用的流程为：

- ◎ 初始化一个 TensorFlowInferenceInterface 实例，并在构造函数中传入 TensorFlow 模型文件名。
- ◎ 调用该实例的 feed()方法，传入输入层特征数据。
- ◎ 调用该实例 run()方法，执行推理/预测。
- ◎ 调用该实例 fetch()方法，读取输出层的预测结果。

思考：如果你学过前面几章的 TensorFlow，那么 feed()、run()、fetch()是不是似曾相识？

很明显，该接口的三个方法名称源于 TensorFlow 中执行图结构的术语，只是改造成了同步调用的形式，更符合普通开发者的编码习惯。

11.1.2　iOS 上的 CoreML

虽然 TensorFlow 官方网站的 Mobile 部分也推荐了在 iOS 上实践 TensorFlow 模型的开发指导，但它们还是基于 iOS 9 的软硬件技术，目前已经略显过时。苹果在 2017 年的 iOS

11 中发布的 CoreML 则可以更好地利用最新苹果手机的硬件资源。

1. CoreML Tool 介绍

苹果的开发者网站上有比较丰富的 CoreML 应用案例，其中有与 Android 类似的已训练图像模型和调用源码。此外，苹果还发布了一个 Python 模型转换工具 CoreML Tool，使用它可以将各种机器学习工具训练的模型转换成 CoreML 格式的模型。图 11-2 列出了几种源模型格式。

模型类别	支持源模型格式	支持框架
Neural networks	Feedforward, convolutional, recurrent	Caffe v1 Keras 1.2.2+
Tree ensembles	Random forests, boosted trees, decision trees	scikit-learn 0.18 XGBoost 0.6
Support vector machines	Scalar regression, multiclass classification	scikit-learn 0.18 LIBSVM 3.22
Generalized linear models	Linear regression, logistic regression	scikit-learn 0.18
Feature engineering	Sparse vectorization, dense vectorization, categorical processing	scikit-learn 0.18
Pipeline models	Sequentially chained models	scikit-learn 0.18

图 11-2　CoreML 支持的几种源模型格式

图 11-2 中大多数模型在本书前几章都有详细分析，这里不再解释。有了这种移植各种模型的能力，现在 iOS 几乎能够胜任所有机器学习技术。

注意：CoreML 针对 TensorFlow 的模型转换工具是 TFCoreML，使用方式与 CoreML Tool 类似，本书不再重复。

2. CoreML Tool 的使用

CoreML Tool 在使用方面非常简单，首先用 pip3 命令安装该组件：

```
# pip3 install -U keras                          # CoreML Tool 依赖于 Keras
# pip install -U coremltools
```

然后就可以开始模型转换了。以 scikit-learn 为例，通过几行代码就可以将一个 scikit-Learn 模型转换为能在 iOS 上使用的 CoreML 模型文件。

```
from sklearn.linear_model import LinearRegression  # 某个 scikit-learn 模型

model = LinearRegression()
model.fit([[1, 2], [3, 4]], [0, 1])                # 训练

import coremltools                                 # 引入 CoreML Tool
coreml_model = coremltools.converters.sklearn.convert( # 转换为 CoreML 模型
   model, ["feature1", "feature2"], "label")       # 给出特征与标签的名称

coreml_model.save('sample.mlmodel')                # 保存到文件
```

其他几种框架的转换方式类似，都是用形如 coremltools.converters.XXX.convert 的函数转换源模型即可。

3. 开发简介

用 Xcode 开发的 iOS App 提供了以“傻瓜”方式调用 CoreML 机器学习模型的方法：将*.mlmodel 文件用鼠标拖到 Xcode 中，即可自动生成 Swift 或 Objective-C 的模型调用类，如图 11-3 所示。

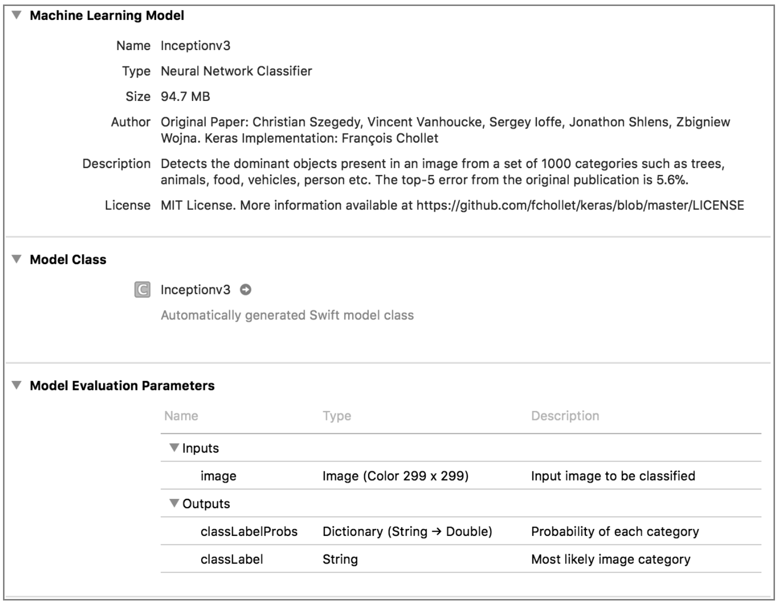


图 11-3　将 Xcode 导入 CoreML 模型

图 11-3 中清楚地描述了模型基本信息、模型封装类的名字（Inceptionv3）、输入与输

出数据格式。在需要用该模型预测的代码中，实例化一个封装类对象，然后调用该对象的 prediction()函数即可。

11.2　迁移学习

与上一节完全复用已有模型的“迁移”不同，本节讨论的是在已有模型的基础上开发出新的模型，这就是所谓的迁移学习（Transfer Learning）。

思考：迁移学习与机器学习的不同？

11.2.1　动机

如本书第 1 章介绍的那样，在机器学习应用的整个研发周期中，模型的反复训练、验证与调试是最耗时的工作，这一点读者在后面的实践中也会有深刻的体会。随着机器学习应用的不断增多，人们尝试避免在每个项目中都重复这样类似的工作，希望能把类似场景的模型在一定程度上复用起来。

另一方面，从零开始训练模型需要大量的样本数据，这在图像、自然语言等超高维特征场景中显得更为突出。很多特定领域的标签标注工作需要按样本条数计人工成本，这在工业界被戏称为“人工智能：有多少人工，就有多少智能”。

由于训练时间、数据成本两方面的约束，使得迁移学习受到越来越多的关注，它能在利用已有模型的基础上，仅利用有限的样本数据训练出泛化能力不错的新模型。这是如何实现的呢？如图 11-4 所示给出了神经网络模型下迁移学习的实现原理。

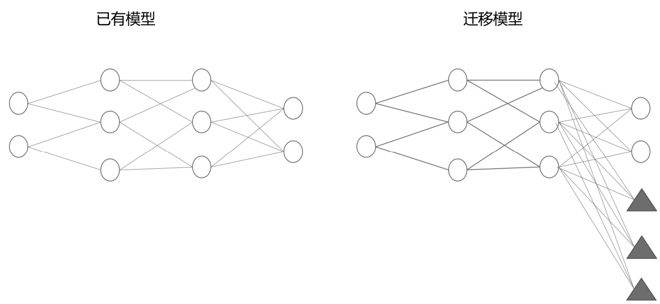


图 11-4　神经网络模型下迁移学习的实现原理

在图 11-4 中，假设已有一个训练好的 4 层网络模型用于区分“苹果、香蕉”两种水果。现在有一个需求，希望新建一个模型能够对“苹果、香蕉、葡萄”三种水果进行分类。按照之前的理论，此时需要重新利用所有样本训练和调试出一个模型。但如果利用了迁移学习，就可以保留原来网络模型中的输入层和隐藏层，转而用少量的三种水果的样本训练一组新的输出层。即，在图 11-4 右侧的迁移模型中，只需对实心结点的相关变量进行反向传播梯度优化。

为什么可以这样做呢？回想第 9 章中介绍的卷积层物理意义，它像一个放大镜逐层地在原始图像中搜寻泛化特征，因此可以认为每个卷积层是已经学习到了原始数据中的某种知识。那么在这些隐藏层上再进一步学习其他知识，要比从原始数据上学来得容易。

这就好比让一个数学系的研究生转作机器学习研究，比让一个初中生去做容易得多，因为数学系的研究生可以在已有高等数学知识（即隐藏层）的基础上学习新的技术。

但必须要认识到的是，迁移学习在实现上有比上述动机理解方面更困难的地方，有更多的工作要做。比如对原模型有哪些量化的检验标准、选择在网络的哪一层开始重新训练，都是迁移学习作为一个研究方向正在探索的问题。

11.2.2 训练流程

迁移学习动机与原理非常易于理解，但如何在项目中实践迁移学习开发并不是那么一目了然。其实，相比从零开始的机器学习模型，训练迁移学习模型只是多了一个样本生成的过程。如图 11-5 所示是关于如何训练图 11-4 所示迁移学习模型的开发流程图。

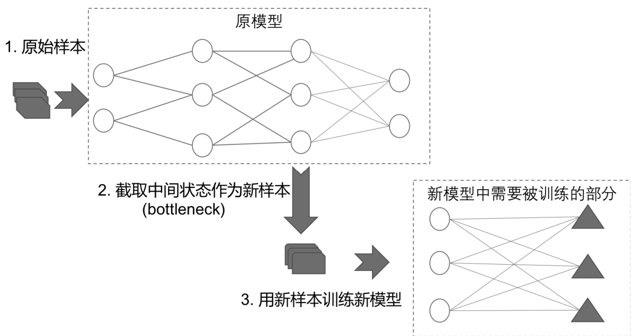


图 11-5 迁移学习训练流程图

在图 11-5 中，“原模型”是已经被训练好的被迁移模型，要利用它在新的“原始样本”

上训练出新模型。该过程为：

（1）将原始样本传入被迁移模型执行前向预测。

（2）在预测过程中，从迁移层截取所有样本在该层的中间状态，作为新的样本保存在内存或物理文件中。这种被截取的新样本有一个术语名称——bottleneck。

（3）用 bottleneck 样本训练新模型。

第（3）步中新模型的训练过程就和普通的神经网络完全一样了，也包含对样本数据的 shuffle、repeat 等操作，然后进行模型拟合、验证、调参迭代等。训练完成后、将新训练的层次接续在原模型的 bottleneck 层之后，即可得到完整的新模型。

11.3　案例实战：基于 TensorFlow Hub 的迁移学习开发

在 2018 年的 TensorFlow 开发峰会上，一个最大的焦点就是 TensorFlow Hub 的正式发布。它是一个可重用 TensorFlow 模型库，其中收纳了很多被训练良好的语言模型与图像识别模型，其目的就是允许使用者基于这些基础模型进行迁移学习开发。

本节解析 TensorFlow Hub 官网上的花卉识别项目，指导读者完成迁移学习的实战开发。在实践之前，通过 pip3 命令安装 TensorFlow Hub：

```
# pip3 install tensorflow-hub
```

11.3.1　下载并训练

如图 11-5 所示，迁移学习开发的前提是具备：

◎　训练样本。

◎　原模型。

在基于 TensorFlow Hub 的迁移学习开发中，前者需要开发者自己准备（或者从 TensorFlow 网站下载示例样本），后者可以通过 TensorFlow Hub 所提供的 API 从 GitHub 上下载。从官网下载本项目训练样本的方法如下。

```
# curl -LO http://download.tensorflow.org/example_images/flower_photos.tgz
```

```
# tar -zxvf flower_photos.tgz
```

然后下载示例迁移学习训练程序并运行：

```
# curl -LO https://github.com/tensorflow/hub/raw/r0.1/examples/image_retraining/retrain.py
# python3 retrain.py --image_dir ./flower_photos
```

该程序首先自动从官网下载一个 inception_v3 花卉识别已训练好的模型，然后利用该模型在 flower_photos 上进行迁移学习训练。其中 flower_photos 中的样本不包括在训练原始 inception_v3 模型的样本中。

11.3.2 检验学习成果

在训练结束后，新模型保存在/tmp/output_graph.pb 文件中。可以下载官方 GitHub 上的另外一个程序，检验新模型对花卉的识别能力。比如：

```
# curl -LO https://github.com/tensorflow/tensorflow/raw/master/tensorflow/examples/label_image/ label_image.py
# python3 label_image.py --graph=/tmp/output_graph.pb --labels=/tmp/output_labels.txt --input_layer=Placeholder --output_layer=final_result --image=./flower_photos/roses/12240303_80d87f77a3_n.jpg
```

其中 image 参数指定的是要识别图像文件的名称。这里传入的仍然是下载的 flower_photos 样本集中的文件，该样本集的内容如图 11-6 所示。

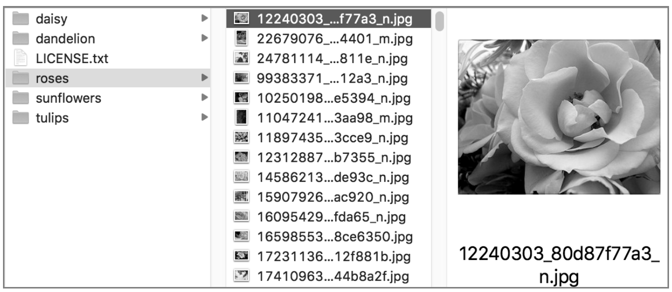


图 11-6　flower_photos 样本集

对于图 11-6 中所示图片给出的预测结果为：

```
roses 0.9862179
tulips 0.013723418
sunflowers 3.7595066e-05
```

```
daisy 1.3961036e-05
dandelion 7.034586e-06
```

即模型认为超过 98%的可能是：该图片为玫瑰。显然效果非常不错。

11.3.3　迁移学习开发

本案例的模型分类程序 label_image.py 与第 9 章内容相比没有新知识点，留给读者自己分析。接下来讲解 retrain.py 中用于迁移学习的关键代码：

（1）通过 hub.load_module_spec()可以下载 TensorFlow Hub 上的已训练模型，比如：

```
import tensorflow_hub as hub

module_spec = hub.load_module_spec(FLAGS.tfhub_module)
graph, bottleneck_tensor, resized_image_tensor, wants_quantization = (
     create_module_graph(module_spec))
```

其中 tfhub_module 是形如'https://tfhub.dev/google/imagenet/inception_v3/feature_vector/1'的模型地址。create_module_graph() 函数从下载的模型中提取出图模型、bottleneck 层等变量。

思考：还记得 bottleneck 层是指什么吗？

（2）用得到的 bottleneck_tensor 执行图结构，可以得到图 11-5 中 bottleneck 层产生的“新样本”：

```
  resized_input_values = sess.run(decoded_image_tensor,    # 转换图像大小
                              {image_data_tensor: image_data})
  bottleneck_values = sess.run(bottleneck_tensor,  # 读取 bottleneck 样本
                           {resized_input_tensor: resized_input_values})
```

（3）新建一个分类层，接续在 bottleneck 层之后：

```
with graph.as_default():                                # 使用原模型图结构
    (train_step, cross_entropy, bottleneck_input,
     ground_truth_input, final_tensor) = add_final_retrain_ops(# 追加一层
         class_count, FLAGS.final_tensor_name, bottleneck_tensor,
         wants_quantization, is_training=True)
```

这样，新模型的最终输出张量是 final_tensor。

（4）用 bottleneck 样本训练新模型：

```
train_accuracy, cross_entropy_value = sess.run(
    [evaluation_step, cross_entropy],
    feed_dict={bottleneck_input: train_bottlenecks,   # bottleneck 样本
               ground_truth_input: train_ground_truth}) # 样本标签
```

代码文件 retrain.py 中的代码有逾千行，不再详细列举。读者在阅读时只需要抓住这里列出的关键点，就可以很容易地掌握迁移学习的开发技巧。

11.4 本章内容回顾

谷歌在 Android 上发布了很多已训练的图像/文本深度学习模型，非常便于集成；苹果在 iOS 11 及其后续版本中也制作了很多已训练模型。

CoreML Tool 可以将用 TensorFlow、scikit-learn 训练的模型转换为在 iOS 上可以直接使用的模型。

迁移学习是一种基于已训练模型、通过较少样本即可训练出泛化能力较强新模型的机器学习方式。

迁移学习模型的训练方式为，先通过已有模型生成 bottleneck 样本，然后用 bottleneck 样本训练新模型。

在 TensorFlow Hub 上发布了很多用于迁移学习的基础模型，可查询官方网站获取最新模型信息。

后　记

在本书的写作中，笔者尽力恪守在前言中所描述的目标，围绕算法意图与实践应用讲解每一个模型。一些模型可以直接参考原始论文或经典总结，更多的模型则通过回顾多个论文与应用、融汇筛选顶尖专家们的研究成果而成文。

这最终成为一个在整体与细节之间进行权衡的过程。大多数初学者感兴趣的可能是整体问题，比如决策树与贝叶斯模型的选择、如何理解流行学习、Bernoulli/Beta/Multinomial/Dirichlet 等常见概率模型的数学含义、深度学习通过哪些手段解决梯度消失问题、支撑自然语言处理和人机博弈等智能应用的实践原理等，针对这些问题书中都进行了较全面的论述。

随着学习的深入，读者可能会不满足于这些话题，并对更多的算法细节与模型分支产生兴趣，比如决策树算法的三种重要实现、Gibbs 采样有效性或变分贝叶斯的数学基础、深度学习中使用不同优化器对模型产生的影响、追求更高的图像识别或棋类博弈的智能水平等。它们就像是从机器学习这棵大树下不断生长与裂变的根茎，越向深处探索越难以拥抱所有。探索这些更细节领域的最好方式不再是求全责备，而是选择一个方向研究下去，按图索骥、蔓引株连。在这方面，本书多数章节都阐述了笔者对于如何在该领域内进行更深度研究的想法，并列出经典论文或实践案例。

从读书经历来说，笔者认为，不能让读者在一个月内读完的书都称不上是好书（当然，不包括资本论、相对论、道德经）。除在深度方面不得不选择合适的程度进行取舍外，在广度方面也对知识进行了适当剔除，比如异常与边缘检测、几乎所有数据挖掘书籍都会讲

解的 Apriori 关联算法、近两年获得较多关注的 GAN 生成对抗网络等。没有加入这些知识，一方面是因为较难将它们归类到本书的主要知识框架中，另一方面则是因为完成本书内容的学习后已经扫清了对这些知识的阅读障碍，读者可以凭借兴趣在互联网上学习相关内容。

这个时代的人工智能技术追随者是幸运的，记得自己在十几年前刚开始接触机器学习时只能从 www.codeproject.com 上寻找神经网络的 C++版本开源实践案例，而现在则有如此丰富易用的编程语言与成熟框架。关于书中的实践内容，它们的原型有的来自工具的官方教程，有的来自在 GitHub 上关注度较高的开源项目。书中对它们都做了些调整，目的除了应用书中的知识点，还希望能起到鼓励和锻炼读者站在巨人的肩膀上、将经典实现案例快速应用到项目中的能力。

如果读者在读完本书后感到意犹未尽，对书中未讲述的模型能够触类旁通，有兴趣直接阅读未来人工智能领域的新论文，那将是笔者完成这本书的最大回报！

刘长龙

电子工业出版社 PUBLISHING HOUSE OF ELECTRONICS INDUSTRY http://www.phei.com.cn Broadview® WWW.BROADVIEW.COM.CN

博文视点 · IT出版旗舰品牌

博文视点诚邀精锐作者加盟

十载耕耘奠定专业地位

以书为证彰显卓越品质

《C++Primer（中文版）（第5版）》、《淘宝技术这十年》、《代码大全》、《Windows内核情景分析》、《加密与解密》、《编程之美》、《VC++深入详解》、《SEO实战密码》、《PPT演义》……

“圣经”级图书光耀夺目，被无数读者朋友奉为案头手册传世经典。

潘爱民、毛德操、张亚勤、张宏江、昝辉Zac、李刚、曹江华……

“明星”级作者济济一堂，他们的名字熠熠生辉，与IT业的蓬勃发展紧密相连。

十年的开拓、探索和励精图治，成就**博**古通今、**文**圆质方、**视**角独特、**点**石成金之计算机图书的风向标杆：博文视点。

“凤翱翔于千仞兮，非梧不栖”，博文视点欢迎更多才华横溢、锐意创新的作者朋友加盟，与大师并列于IT专业出版之巅。

英雄帖

江湖风云起，代有才人出。
IT界群雄并起，逐鹿中原。
博文视点诚邀天下技术英豪加入，
指点江山，激扬文字
传播信息技术，分享IT心得

• 专业的作者服务 •

博文视点自成立以来一直专注于 IT 专业技术图书的出版，拥有丰富的与技术图书作者合作的经验，并参照IT技术图书的特点，打造了一支高效运转、富有服务意识的编辑出版团队。我们始终坚持：

善待作者——我们会把出版流程整理得清晰简明，为作者提供优厚的稿酬服务，解除作者的顾虑，安心写作，展现出最好的作品。

尊重作者——我们尊重每一位作者的技术实力和生活习惯，并会参照作者实际的工作、生活节奏，量身制定写作计划，确保合作顺利进行。

提升作者——我们打造精品图书，更要打造知名作者。博文视点致力于通过图书提升作者的个人品牌和技术影响力，为作者的事业开拓带来更多的机会。



联系我们

博文视点官网：http://www.broadview.com.cn
CSDN官方博客：http://blog.csdn.net/broadview2006/
投稿电话：010-51260888　88254368
投稿邮箱：jsj@phei.com.cn

新浪微博 weibo.com @博文视点Broadview

微信公众账号 博文视点Broadview

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：（010）88254396；（010）88258888

传　　真：（010）88254397

E－mail：　dbqq@phei.com.cn

通信地址：北京市万寿路173信箱　电子工业出版社总编办公室

邮　　编：100036